A HIGHLY FUNCTIONAL DECISION PARADIGM BASED ON NONLINEAR ADAPTIVE GENETIC ALGORITHM

Final Report

Contract No. DAAH04-93-C-0047

Period of Performance: 08-10-93 to 02-09-94

Sequence No. 0002

Sponsor:

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

Technical Monitor:

Dr. William Sander
(919) 549-4316

Contractor:

Physical Optics Corporation
Applied Technology Division
2545 West 237th Street, Suite B
Torrance, CA 90505

Principal Investigator:

Dai Hyun Kim, Ph.D.
(310) 530-1416

April 20, 1994

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 9, 1994	Final 8/10/93 – 2/9/94

4. TITLE AND SUBTITLE
A Highly Functional Decision Paradigm Based on Nonlinear Adaptive Genetic Algorithm

5. FUNDING NUMBERS
DAAH04-93-C-0047

6. AUTHOR(S)
Dai Hyun Kim

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Physical Optics Corporation
20600 Gramercy Place, Building 100
Torrance, California 90501

8. PERFORMING ORGANIZATION REPORT NUMBER
3237

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Contracting Officer
U.S. Army Research Office
P. O. Box 12211
Research Triangle Park, NC 27704-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
ARO 31432.1-EL-SBI

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In response to DOD Army solicitation A93-023, POC proposed a new type of processing module that is far superior to any other type of computing method. The greatest advantage of POC's processing module is that it combines the ability of a human to adapt to changing facts with the constant, uniform behavior, predictability, speed, large information storage capacity and non-emotional responses of machines. This proposed highly dimensional decision module is based on the genetic algorithm (GA). In Phase I of this program, POC successfully demonstrated a universal decision making method based on GAs and fuzzy rules. The algorithm of this decision making method can be used for multi-sensor feature space data, high dimensionality, and dynamically changing environments. In addition, the algorithm provides the capability to build a nonlinear adaptive system that incorporates many of the properties of human cognitive methods. The multidimensional (10 dimensions) function demonstration of the optimization algorithm in Phase I has fully proved the feasibility of POC's high dimensionality decision making approach. Based on these results, POC believes that the high dimensionality decision module can be fully implemented with existing optical or electronic components in Phase II.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Multidimensional Decision Making, Genetic Algorithms, Fuzzy Rules	70
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 INTRODUCTION

The fusion of remotely sensed data has long been used to derive more information from a given source. The application of multiple sensors to the problems of detection, tracking and identification offer numerous potential performance benefits over traditional single-sensor approaches[1-3]. The characteristics of multi-sensor systems that can provide operational benefits to specific applications include robust operational performance, extended spatial coverage, extended temporal coverage, increased confidence, reduced ambiguity, improved detection performance, enhanced spatial resolution, and improved system operational reliability[4-5]. All of these benefits can be realized by one very important factor: a high dimensionality decision making capability. The measurement space of a multi-sensored system has a very high dimensionality because of the very fact that it uses multiple sensors. Therefore, it is essential that the decision module analyze a *high dimensionality feature space* with dynamic adaptation of changing environments. Such a decision making system will be extremely valuable for fast, reliable performance under complex circumstances.

In response to DOD Army solicitation A93-023, POC proposed a new type of processing module that is far superior to any other type of computing method. The greatest advantage of POC's processing module is that it combines the ability of a human to adapt to changing facts with the constant, uniform behavior, predictability, speed, large information storage capacity and non-emotional responses of machines. This proposed highly dimensional decision module is based on the *genetic algorithm* (GA) with the adaptation of *fuzzy rules*, which control the processes of each generation more efficiently when compared to existing conventional GAs.

The GA, developed by Holland[6], studies the adaptive process of natural biological systems and develops artificial systems that mimic their adaptive mechanism. Recently, genetic algorithms have been successfully applied to various optimization problems, such as the traveling salesman problem[7-8], image processing[9-10], neural network training [11-13], and other search and optimization operations[14-16]. Genetic algorithms differ from traditional optimization methods, in that they combine the concept of "survival of the fittest" among strong structures with a structured, yet randomized, information exchange. The result is a search algorithm that possesses some of the innovative flare of human intuition when applied to problem solving. In each generation of the GA, a new set of strings is created using bits and pieces of the fittest of the previous generation. The advantages of GAs are as follows:

1. Genetic algorithms use a coding of the parameter set rather than the parameters themselves.
2. Genetic algorithms search from a population of search nodes instead of from a single one.
3. Genetic algorithms use probabilistic transition rules.
4. Genetic algorithms use objective function information instead of derivatives or other auxiliary knowledge, which are usually tedious to perform.

A genetic algorithm consists of a string representation ("genes") of the nodes in the search space, a set of genetic operators for generating new search nodes, a fitness function to evaluate the search nodes, and a stochastic assignment to control the genetic operators. Here, each gene presents each sensor with a level of dimension.

Typically, a genetic algorithm consists of the following steps.

1. Initialization—an initial population of the search nodes is randomly generated.
2. Evaluation of the fitness function—the fitness value of each node is calculated according to the fitness function (objective function).
3. Genetic operations—new search nodes are generated randomly by examining the fitness value of the search nodes and applying the genetic operators to the search nodes.
4. Repeat steps 2 and 3 until the algorithm converges.

From the above description, it can be seen that genetic algorithms use the notion of "survival of the fittest" by passing "good" genes to the next generation of strings, and by combining different strings to explore new search points. The construction of a genetic algorithm for any problem can be separated into four distinct, yet related tasks.

1. The choice of the representation of the strings,
2. The design of the genetic operators,
3. The determination of the fitness function, and
4. The determination of the probabilities controlling the genetic operators.

In this Phase I program, POC focused on determining the above tasks. The results of these determinations are presented in Section 3.

1.1 High Dimensionality Decision Capability of GAs

The mechanics of GAs are very simple, and involve generating, copying and swapping strings. Each string is a binary representation of parameters. Each parameter has a different dimensional factor. For example, a four dimensional pattern representing a *chromosome* (string) consists of four *genes*, as shown in Figure 1-1.

Figure 1-1.
Control Parameters Represented as a Chromosome.

In order to increase the dimensionality, one can simply increase the number of genes in each string, thus changing the fitness function. Therefore, using GAs high dimensionality decision making is very simple, when compared to any other traditional optimization method. In each gene, the number of binary bits can be of different lengths. Thus, in the case of multi-sensor fusion, the number of binary bits for each gene can be chosen depending on the resolution of each sensor. This flexibility provides very high adaptability to varying environments, which is the most important issue for high dimensionality decision making.

1.2 Fundamental Procedure of GAs

Basically, GAs are divided into two main operations; random initialization and evolution, as shown by the computer program in Figure 1-2.

```
Random Initialization:      initialize ();
                            evaluate ();
                            select ()
Evolution Procedure:        for (1=0:1<MAX-GENERATIONS;++i) {
                            reproduce ();
                            crossover ();
                            mutate ();
                            evaluate ()
                            }
```

Figure 1-2.
Two Main Genetic Algorithm Operations; Random Initialization and Evolution.

In the initialization function, a population of chromosomes is created by randomly selecting values for the genes. Each variable is allowed to span its own fixed domain of floating points (or discrete values); therefore, the chromosome may be composed of genes spanning continuous and/or discrete domains. This is permissible, provided that the gene locations corresponding to each variable remain intact. Next, each chromosome is evaluated for its fitness in the evaluation module. The fitness is related to the cost function, in that the best estimates will have the largest fitness values. Once the chromosomes have been assigned values, parent chromosomes for the next generation are selected.

In the first step, the parent chromosomes are reproduced. Next, a probability is assigned to each chromosome, according to a fixed probability distribution. Although this is a violation of the schema theorem, it circumvents the difficulties associated with fitness scaling, and provides a more general algorithm for arbitrary cost function mappings. Finally, the selection process is performed in pairs using a random probability selection weighted by the aforementioned function. This ensures that copies of the strongest chromosomes are passed on to the next generation. Once a pair has been selected, it is permitted to breed and form a pair in the new generation. In this procedure, there are three major evolution methods: *reproduction*, *crossover* and *mutation*.

The role of each evolution is as follows:

1. Reproduction copies chromosomes from one generation to the next using their relative fitness values.

2. Crossover chooses a random location in two chromosomes. Then, the chromosomes are crossed so that the resulting pair contains material from both of the two parent chromosomes.

3. Mutation flips the value of a randomly selected bit or a set of bits.

There are many methods that can be used to implement each evolution procedure. In this program, POC determined and implemented the most effective and fastest mutation and crossover algorithm, which is based on fuzzy logic controller that is suitable for high dimensionality decision making. Since in each generation most of the new breed of chromosomes are redundant and rebred, the fuzzy logic controller is used to reduce the number of iterations, and thus overall optimization process. Also, it is a commonly held misconception that crossover is the most important function. In our new discovery, however, it was found that mutation is the most important. In particular, when we deal with the Rosenbrock's function (also known as the *banana function*), mutation is an essential step to optimize the function. In Table 1-1, a comparison of POC's new adaptive

GA to other GAs is shown. For this comparison, we used a sorting problem for 2D data space, with each dimension in the image from 0 to 2^{20}. This comparison was done using a 66 MHz IBM compatible computer. Much higher performance can be obtained by using such powerful computers as Cray-MP or Touchstone Delta. Even with the more powerful computers, however, the higher performance of the FEGA will still be evident.

Table 1-1. Excitation Time for Different Sorting Algorithms

Algorithm Type	Excitation Time
Bubble Sorting	1.13×10^9 hours
CARM Sorting	14.4 hours
Conventional GA	2 – 3 seconds
POC FEGA	0.3 – 0.5 seconds

2.0 HIGHLIGHTS OF PHASE I TECHNICAL OBJECTIVES AND RESULTS

In Phase I of this program, POC successfully demonstrated a universal decision making method based on GAs and fuzzy rules. The algorithm of this decision making method can be used for multi-sensor feature space data, high dimensionality, and dynamically changing environments. In addition, the algorithm provides the capability to build a nonlinear adaptive system that incorporates many of the properties of human cognitive methods.

The specific objectives of Phase I were: (1) design the parallel neural network evolution hardware (genetic evolvers); (2) simulation of such system on a digital computer; (3) perform a parametric study of speed and population size; and (4) design the Phase II prototype. POC successfully completed the tasks as follows:

1. Designed the parallel genetic algorithm evolvers,
2. Performed the computer simulation,
3. Determined the speed and population size,
4. Developed the most efficient mutation method, and
5. Determined the adaptability of fuzzy rules to GAs.

The multidimensional (10 dimensions) function demonstration of the optimization algorithm in Phase I has fully proved the feasibility of POC's high dimensionality decision making approach. In fact, the accuracy of this method for the banana function even exceeded our expectations. Based on these results, POC believes that the high dimensionality decision module can be fully implemented with existing optical or electronic components in Phase II.

The key achievements which fully satisfy the Phase I objectives are:

1. The demonstration of high dimensionality decision making. A 10-dimensional function with 2^{20} dynamic range for each gene was successfuliy obtained.

2. The determination of fuzzy rules. Using fuzzy rules, unnecessary mutations (for early iterations) and crossover (for later iterations) can be

successfully eliminated. Therefore, the overall computing speed can be increased by at least one order of magnitude in comparison to other GA algorithms.

3. The reevaluation of the mutation. The importance of the mutation is well-defined in this program. POC proved that a GA with only reproduction and mutation can be used, while a GA with only reproduction and crossover cannot be used. Therefore, in the Phase I prototype implementation, the overall system architecture can be minimized.

4. The design of the GA evolver. Optical crossover and mutation evolvers were designed using only spatial modulators, prisms, and detectors (see Monthly Report for period 8/93-9/93).

5. The design of the Phase II prototype. A tentative Phase II prototype has been designed. This design will be optimized during the first stage of Phase II program.

3.0 PHASE I RESULTS

In this Phase I program, POC focused on the following three major parts: (1) Development of practical GA evolution algorithms (such as reproduction, crossover and mutation), and computer simulation; (2) Fuzzy logic implementation and application to GAs; and (3) Design of the optical/electronic GA evolvers. The overall computer program structure of the GA is shown in Figure 3-1. In the next section, a more detailed description and explanation is included.

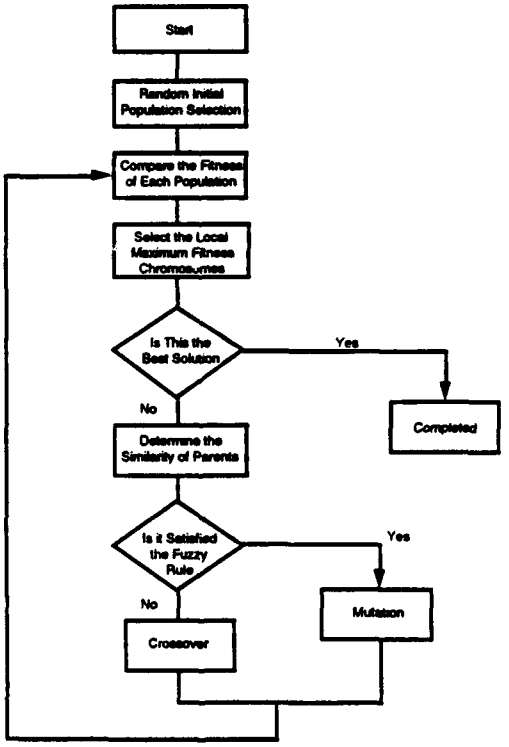

Figure 3-1.
Flow Chart of General GA.

3.1 Description of GA Computer Simulation

GAs are powerful search algorithms based on the mechanics of natural genetics. They ensure the proliferation of quality solutions while investigating new solutions via a systematic information exchange that utilizes probabilistic decisions. It is this combination that enables GAs to exploit historical information to locate new points in the search space with expected improved performance.

GAs are unlike many conventional search algorithms in the following ways:

1. GAs consider many points in the search space simultaneously, not a single point;
2. GAs work directly with strings of characters representing the parameter set, not the parameters themselves;
3. GAs use probabilistic rules to guide their search, not deterministic rules.

GAs consider many points in the search space simultaneously, and therefore have a reduced chance of converging to local optima. In most conventional search techniques, a single point is considered based on some decision rule. These methods can be dangerous in multimodal (many peaked) search spaces; because they can converge to local optima. However, GAs generate entire populations of points (coded strings), test each point independently, and combine qualities from existing points to form a new population containing improved points. Aside from producing a more global search, the GAs simultaneous consideration of many points makes it highly adaptable to parallel processors, since the evaluation of each point requires independent computation.

GAs require the natural parameter set of the problem to be coded as a finite length string of characters. The parameter sets in this study, which consist of an entire set of fuzzy membership functions describing both condition and action variables, are coded as strings of "0" and "1". For example, one of the parameters considered in this study is a point that defines the upper limit of a particular trapezoidal membership function, P, as shown in Figure 3-2. Note that four points are required to completely define a single trapezoidal membership function. This parameter may easily be represented as a binary string. Seven bits are allotted for defining each of the four points needed to completely define each trapezoidal membership function. The choice of seven bits was not arbitrary· in the current problem, seven bits allowed for adequate "resolution" while preventing the problem from becoming too large to comfortably handle. These seven bits are interpreted as a binary number (0000011 being the binary number 3). This value is mapped linearly between user-determined minimum (P_{min}) and maximum (P_{max}) values according to the following relation:

$$P = P_{min} + \frac{b}{\left(2^m - 1\right)}\left(P_{max} - P_{min}\right) \qquad (1)$$

where P is the value of the parameter being coded and b is the integer value represented by an m bit string.

This coding technique provides a convenient means of representing a single parameter: one of four points needed to define a single membership function. (Four points are needed to represent each membership function when trapezoidal membership functions are used, while only three points are needed to represent individual triangular membership functions.) However, each string must represent an entire set of membership functions,

with each membership function describing all of the condition and action variables. This multiple parameter coding can be accomplished by coding each of the necessary parameters in the manner described above (the string lengths for individual parameters may vary), and by concatenating the bit strings.

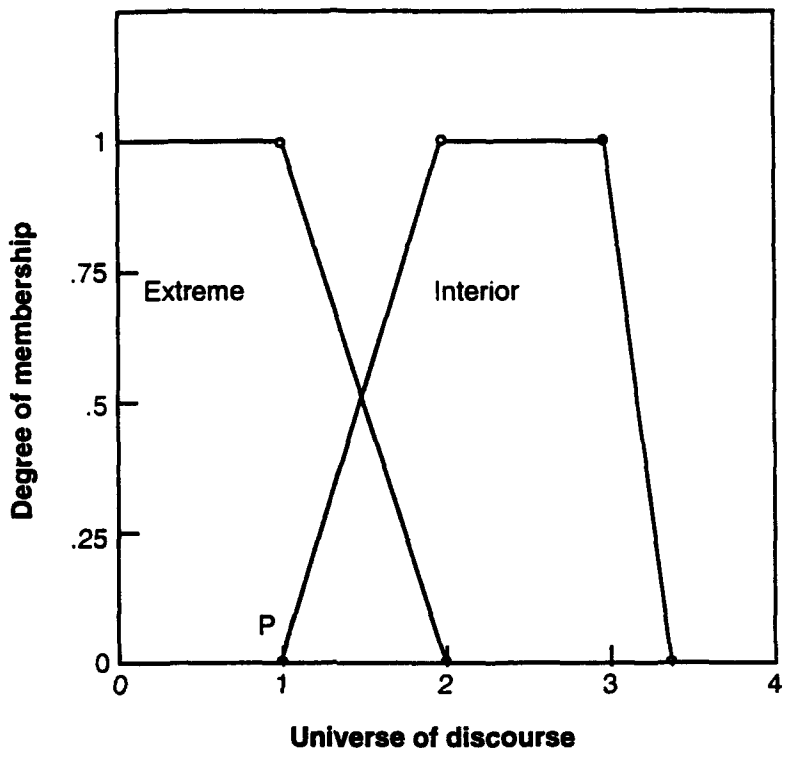

Figure 3-2.
Four points such as the one marked P must be defined for each trapezoidal membership function describing both condition and action variables.

3.1.1 String Representation and Initial Population

This section introduces the string representation used for multidimensional decision making and presents a method to generate an initial population.

3.1.1.1 String Representation

An important factor in selecting the string representation for the search nodes is that all of the search nodes in a search space are represented, and that the representation is unique. It is also desirable, though not necessary, that the strings be in one-to-one correspondence with the search nodes. That is, each string corresponds to a legal search node (see Figure 3-3). The design of the genetic operator is greatly simplified if the string representation and search spaces are in one-to-one correspondence. Consider here the problem of finding a representation for genetic algorithms in the problem of job shop scheduling. An intermediary encoded representation of the schedules and a decoder was used that would always yield legal solutions to the problem. The representation is somewhat complicated and is for a different problem.

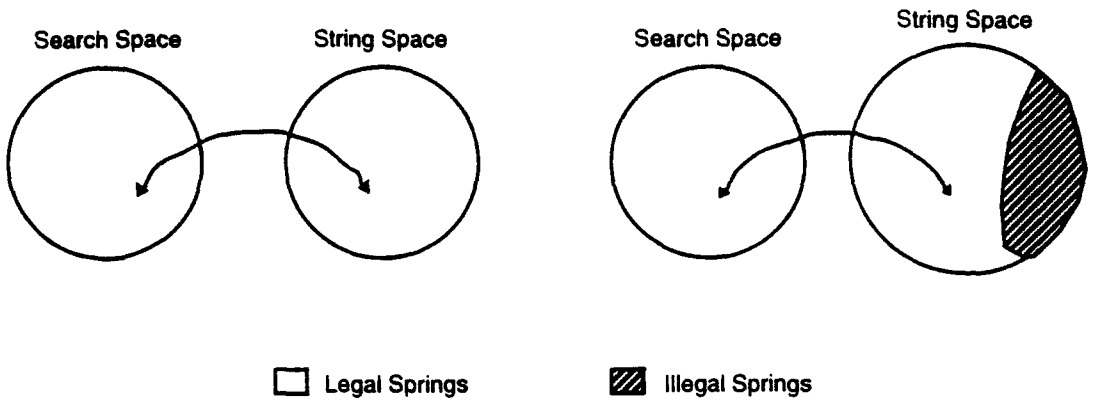

Figure 3-3.
Mapping Between String Representation Space and Search Space.

For the multiprocessor scheduling problem[17], a legal search node (a schedule) is one that satisfies the following conditions.

1. The precedence relations among the tasks are satisfied.
2. Every task is present and appears only once in the schedule (completeness and uniqueness).

The string representation used in this paper is based on the schedule of tasks in each individual sensor or value. This representation eliminates the need to consider the precedence relations between the tasks scheduled to different sensors. The precedence relations within the sensors, however, must still be maintained.

In this Phase I program, a string (chromosome) can be accommodated by up to 20 genes (20 sensors or 20-dimensional functions). Each gene, when 20 genes are used, can be represented by up to a 20 bit string. Therefore, a string can be represented by up to 200 bits. Figure 3-4 shows the string representation.

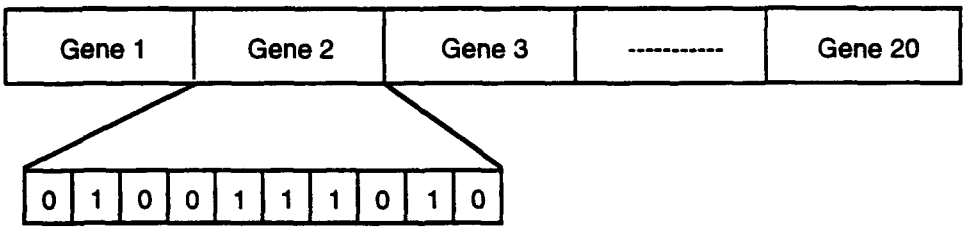

Figure 3-4.
General String Representation of 20 Genes, Each Gene is up to 10-bit.

It should be stressed that each gene can increase its resolution. For example, the dynamic range of a gene is 2^n and its resolution is 2^m, and the number of bits in each gene is n+m. The string representation of the gene is shown in Figure 3-5.

A_0	A_1	A_2	.....	A_{n+m}

Figure 3-5.
String Representation of a Gene What Dynamic Range is 2^n and its Reselection is 2^m.

The value of each allele is:

$$A_0^0 = \left(\frac{1}{2^m}\right) x\, 2^0$$

$$A_1^1 = \left(\frac{1}{2^m}\right) x\, 2^1$$

$$A_2^2 = \left(\frac{1}{2^m}\right) x\, 2^2$$

$$A_{n+m} = \left(\frac{1}{2^m}\right) x\, 2^{n+m-1}$$

As an example, a chromosome which consists of 3 genes, with each gene having a dynamic range of 2^3 and a resolution of 2^2, is represented as a 15-bit string, as shown in Figure 3-6.

A_0	A_1	A_2	A_3	A_4	B_1	B_2	B_3	B_4	B_5	C_1	C_2	C_3	C_4	C_5

Figure 3-6.
String representation of 15-bit chromosome which consists of 3 Genes, dynamic range of each gene is 2^2 its resolution is 2^2.

In the case of 10101 11100 00111, the value of each gene is: for gene A,

$$A = \frac{1}{2^2} x 2^2 + \frac{1}{2^2} x 2^2 + \frac{1}{2^2} x 2^4 = 5.25,$$

$$B = \left(\frac{1}{2^2}\right)\left(2^0 + 2^1 + 2^2\right) = 1.75,$$

$$C = \left(\frac{1}{2^2}\right)\left(2^3 + 2^4\right) = 6.$$

The first major portion of our GA computer program is shown in Figure 3-7 . This part of the computer program randomly generates decimal numbers that are in the range of the gene's dynamic range. Here, chrof [i][j] represents the decimal value of each gene (j^{th} gene of chromosome i). Then, the decimal value of each gene is converted into its corresponding binary string, defined as bit_val[i][k].

```
/* This part of program is to generate a number of randomly selected
   initial genes */

for (i=1; i<=intn; i++)
    {
    for (j=1; j<=ngene; j++)
    {
     chrof[i][j]=0.;
    }
    }

for (i=1; i<=intn; i++)
    {
    for (j=1; j<=ngene; j++)
    {
     int_rand=rand();
     result=ldiv((long)int_rand,mod[j]);
     chrof[i][j]=(double)result.rem;
    }
    }

intm_pxval=0;
for (i=1; i<=intn; i++)
    {
      intm_pxval=0;
      for (j=1; j<=ngene; j++)
      {
       geneval=(int)chrof[i][j];
       for (t=1; t<=nbit[j]; t++)
           {
        k=intm_pxval + t;
        result=ldiv((long)geneval,(long)2);
        bit_val[i][k]=(int)result.rem;
        geneval=(int)result.quot;
        if (t==nbit[j])
           {
            intm_pxval=intm_pxval+nbit[j];
           }
       }

      }
    }
```

Figure 3-7.
Random initial population generation subroutine.

3.1.2 Fitness Function

The fitness function in GAs is basically the objective function that we want to optimize in the problem. It is used to evaluate the search nodes and also controls the generator operators. The fitness functions used for this Phase I program are based on the maximum or minimum value of the multidimensional functions, and are defined as:

$$FT(s) = \chi_{1i}, \chi_{2i}, \cdots \chi_{ni}, f(\chi_{1i}, \chi_{2i}, \cdots \chi_{ni})$$

where $f(\chi_{1i}, \chi_{2i}, \cdots \chi_{ni})$ is the multidimensional function.

In our demonstration computer simulation, five functions are selected, including a 10-dimensional function and a banana function, as shown in Figure 3-8.

```
printf("WHAT FUNCTION DO YOU WANT TO CHOOSE?\n");
printf("\n(1) F1(x,y) = a*(x^b - d)*(y^c - f) + g;\n");
printf("\n(2) F2(x,y) =  (x-2^c)^d + (y-2^g)^h - 2^o\n");
printf("\n(3) F3(x1,x2,x3,x4) = a*x1*x2*x3 - b*x2*x3*x4\n");
printf("\n(4) F4(x1,....,x10) = x1*x2 - x3*x4 + x5*x6 - x7*x8 + x9*x10 \n");
printf("                              - x1*x5*x9\n");
printf("\n(5) Banana function F5(x1, x2) = 562500. - 22500. * x1 + 225.* x1*x1 + \n");
printf(" 9.* x1*x1*x1*x1 - 750. * x1*x1 * x2 + 15625. * x2*x2; \n");
printf("\nTYPE 1, 2, 3, 4 or 5\n");
scanf("%d", &choice);
printf("choice = %d\n" , choice);
```

Figure 3-8.
List of the five fitness functions.

3.2 Genetic Evolvers

GAs are composed of three operators: reproduction, crossover and mutation. These operators are implemented by performing the basic tasks of copying strings (reproduction), exchanging portions of strings (crossover), and by reversing the value of the strings, from 0 to 1 or visa versa (mutation). For the sake of simplicity, we chose two parent chromosomes with 3 genes for each chromosome (see Figure 3-9):

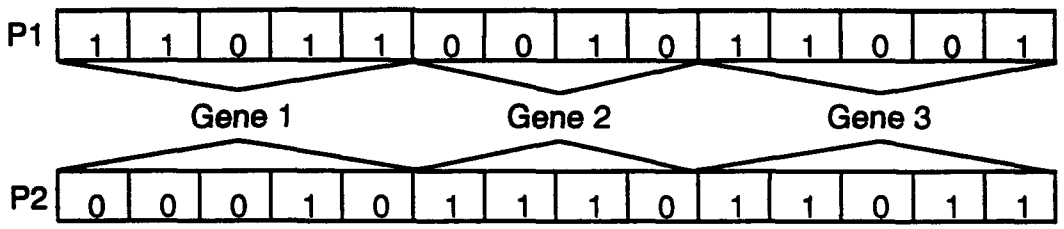

Figure 3-9
Illustration of Two Parent Chromosomes Representing 3 Genes.

3.2.1 Reproduction

Reproduction is a process in which individual strings are copied according to their objective function values (see Figure 3-10). The reproduction operator may be implemented in algorithmic form in a number of ways. POC first selected two maximum and minimum fitness chromosomes as parents from the previous interaction, and then reproduced these maximum and minimum fitness chromosomes. The subroutine of the reproduction was shown in Figure 3-11. In this program, maxparent [j] is the maximum fitness chromosome from the previous iteration, while minparent [j] is the minimum fitness chromosome. The maxparent 2 [j] and the minparent 2 [j] are second maximum and minimum fitness chromosomes, respectively. Therefore, the result of reproduction of p1 and p2 is

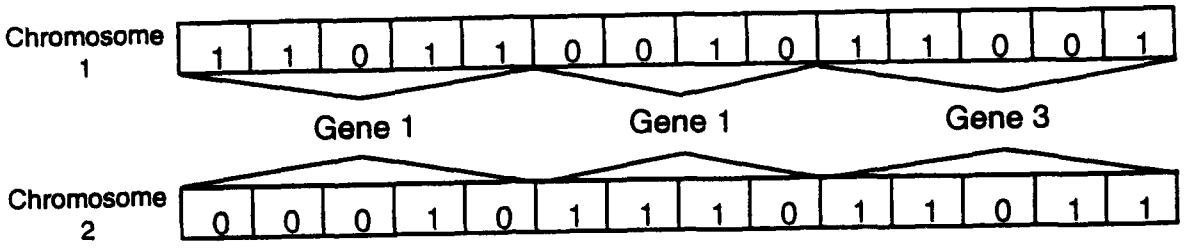

Figure 3-10
Two reproduced chromosomes from the parents

```
void reproduction(void)
{
    /* This part of program is to perform the reproduction of the maximum
       fitness chromesome */
    nb=1;
    ne=4;
    neold=0;
    nbold=0;
    for (j=1; j<=slength; j++)
        {
         bit_val[1][j]=maxparent[j];
         bit_val[2][j]=maxparent2[j];
         bit_val[3][j]=minparent[j];
         bit_val[4][j]=minparent2[j];
         }
       /*   fprintf(fi, "-------reproduction--------\n");
    for (i = nb; i <= ne; i++)
        {
        fprintf(fi, "i = %d ", i);
        for (j = 1; j <= slength; j++)
        {
        fprintf(fi, "%d",bit_val[i][j]);
        if (j==slength) fprintf(fi, "\n");
        }
    }
         printf("The reproduction is finished\n");*/
       nbrep = nb;
       nerep = ne;
       nbold=nb;
       neold=ne;

}
```

Figure 3-11
The subroutine of reproduction program.

3.2.2 Crossover

Crossover exchanges information via probabilistic decisions, and provides a mechanism for strings to mix and match their desirable qualities through a random process. Basically, there are two types of crossover: inter-gene crossover and outer-gene crossover. Inter-

gene crossover performs information exchange within the same gene, while outer-gene performs information exchange between different genes. For example, inter-gene crossover for gene 2 is illustrated in Figure 3-12.

(1) Select a position of strings from each parent (as shown in Figure 3-12).

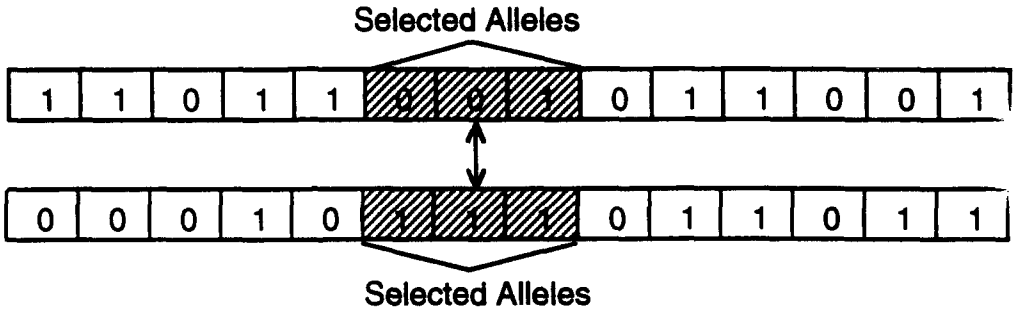

Figure 3-12
Selection of String Position

(2) Exchange the portions of strings to the other parent (the results of inter-gene crossover are shown in Figure 3-13).

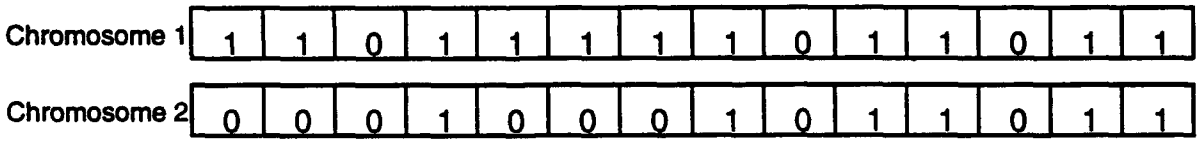

Figure 3-13
Exchange of String Portions

In the case of outer-gene crossover, the procedure is as follows:

(1) Select a portion of strings from any gene, as shown in Figure 3-14 (shaded parts)

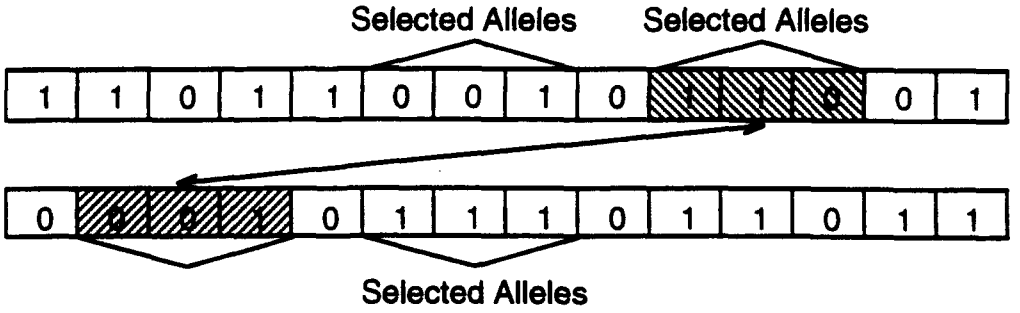

Figure 3-14
Select Portion of Strings

(2) Exchange the portions of strings. The result of outer-gene crossover is shown in Figure 3-15.

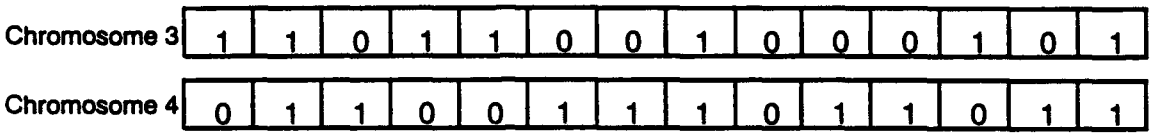

Figure 3-15
Result of Outer-Gene Crossover

The actual number of crossover methods are numerous, and it is still not clear which is the best crossover operation. In this Phase I program, POC learned that the early stages of the iteration outer-crossover speeds the convergence to optimization; while inter-crossover speeds the late stages of the iteration. In Figures 3-16(a) and 3-16(b), the inter-crossover and outer-crossover subroutines are shown.

```
void crossover1(void)
{
    int  k, beg, p, q, pp;
    div_t result;

for (pp = 1; pp<10; pp++)
    {
    result = div(rand(), cross);
    q = result.rem + 1;

    result = div(rand(), slength - q - 1);
    p = result.rem + 1;

    nb = neold+1;
    i = nb;

/*  CROSSOVER OF MAX and MAX2 */

    for (beg = 1; beg <= slength - q + 1; beg++){

        k = p;

        for ( j = 1; j <= slength; j++)

            if ( j < beg || j >= beg + q)
                bit_val[i][j] = maxparent[j];

            else{

                bit_val[i][j] = maxparent2[k];
                k++;
            }

            i++;

            k = beg;
            for ( j = 1; j <= slength; j++)

                if ( j < p || j >= p + q)

                    bit_val[i][j] = maxparent2[j];

                else{

                    bit_val[i][j] = maxparent[k];
                    k++;
                }
            i++;

    }
    ne = i;

    neold = ne;
    nbold = nb;

/*  CROSSOVER OF MIN and MIN2 */

        for (beg = 1; beg <= slength - q + 1; beg++){

        k = p;

        for ( j = 1; j <= slength; j++)

            if ( j < beg || j >= beg + q)

                bit_val[i][j] = minparent[j];
```

Figure 3-16 (a)
Inter-Crossover Subroutine Program

```
                else{

                    bit_val[i][j] = minparent2[k];
                    k++;
                }

                i++;

                k = beg;
                for ( j = 1; j <= slength; j++)

                    if ( j < p || j >= p + q)

                        bit_val[i][j] = minparent2[j];

                    else{

                        bit_val[i][j] = minparent[k];
                        k++;
                    }
                i++;

        }
        ne = i;
}
        neold = ne;
        nbold = nb;

}
```

Figure 3-16 (a)
Inter-Crossover Subroutine Program (continued)

```
void crossover(void)
{
      nb=neold+1;
      ne=neold+80;
      nbcross = nb;
      for (i=nb; i<=ne; i=i+2)
          {
           itt_rand=rand();
           result2=div(itt_rand,cross);
           n=result2.rem+1;
           int_rand=rand();
           result=ldiv((long)int_rand,(long)slength-n-1);
           p=(int)result.rem+1;
           for (j = 0; j < kol; j++)
           if (p == beg[i]){
              if ((p + cbit[i]) > slength)
                 p -= cbit[j];
              else
                 p += cbit[j];
           }

           for (j=1; j<=slength; j++)
           {
             if (p <= j && j < p+n)
                 {
                  bit_val[i][j]=maxparent[j];
                  bit_val[i+1][j]=maxparent2[j];
                 }
              else
                 {
                  bit_val[i][j]=maxparent2[j];
                  bit_val[i+1][j]=maxparent[j];
                 }
           }
           }
      nbold=nb;
      neold=ne;
      nb=neold+1;
      ne=neold+100;

      for (i=nb; i<=ne; i=i+2)
          {
           itt_rand=rand();
           result2=div(itt_rand,cross);
           n=result2.rem+1;
           int_rand=rand();
           result=ldiv((long)int_rand,(long)(slength-n-1));
           p=(int)result.rem+1;
           for (j=1; j<=slength; j++)
           {
             if (p <= j && j < p+n)
                 {
                  bit_val[i][j]=minparent[j];
                  bit_val[i+1][j]=minparent2[j];
                 }
              else
                 {
                  bit_val[i][j]=minparent2[j];
                  bit_val[i+1][j]=minparent[j];
                 }
           }
           }

        nbold=nb;
        neold=ne;
        necross = ne;
   }
```

Figure 3-16 (b).
Outer-Crossover Subroutine Program

3.2.3 Mutation

Mutation can be considered as an occasional random allocation of the values of a string. Until now, mutation was considered to have a secondary role in GAs. However, POC found that mutation is the most important genetic operator among them. To prove this, POC demonstrated searching the minimum values of the banana function, $f(x,y) = 562500 - 22500x + 225x^2 + 9x^4 - 750x^2y + 15625y2$, using only reproduction and crossover, and also using only reproduction and mutation. In the case of the former, the anticipated result (0) could not be obtained after 300 iterations. In the latter case, POC successfully obtained the correct minimum value after only 13 iterations. This provided the most valuable information for designing the Phase I prototype. In this case, the high dimensionality decision module can be implemented using a simple architecture.

As with crossover, there are a number of ways to implement mutations. In this Phase I program, POC selected three essential mutation methods:

(1) Single-bit mutation: Cover only 1 bit from 0 to 1 or visa versa

P1 ⇒ 01101 0110

C1 ⇒ 11101 0110	C6 ⇒ 01101 1110
C2 ⇒ 00101 0110	C7 ⇒ 01101 0010
C3 ⇒ 01001 0110	C8 ⇒ 01101 0100
C4 ⇒ 01111 0110	C9 ⇒ 01101 0111
C5 ⇒ 01100 0110	

(2) Single-gene mutation: This method is to mutate a gene from a single bit to entire bit.

P1 ⇒ 01101 0110

C1 ⇒ 01101 1110	C5 ⇒ 11101 0110
C2 ⇒ 01101 1010	C6 ⇒ 10101 0110
C3 ⇒ 01101 1000	C7 ⇒ 10001 0110
C4 ⇒ 01101 1001	C8 ⇒ 10011 0110
	C9 ⇒ 10010 0110

3. Combination of (1) and (2): In this method, POC combine the method of (1) and (2) that say, P1 is the chromosome of 3 genes.

P1: $\underbrace{011}_{\text{gene1}}\ \underbrace{010}_{\text{gene2}}\ \underbrace{110}_{\text{gene3}}$

In this case, we perform the operation of (2) for gene 3 while perform the operation of (1) for gene 1 and gene 2.

$$C1 \Rightarrow 111110 \begin{cases} 010 \\ 000 \\ 001 \end{cases}$$

$$C2 \Rightarrow 111000 \begin{cases} 010 \\ 000 \\ 001 \end{cases}$$

$$C3 \Rightarrow 111011 \begin{cases} 010 \\ 000 \\ 001 \end{cases}$$
$$\vdots$$

By combining methods (1), (2), and (3), the speed of the convergence of the mutation can be dramatically increased and no crossover is needed to perform the desired optimization. Therefore, the overall system architecture can be extremely simplified and fabricated using existing components.

3.3 Fuzzy Logic and Its Adaptability to GAs

The Genetic Algorithm (GA) is a new approach to the optimizing and searching method. Mostly, the results of GAs are not the optimal solution after many iterations, but only an approximation. During the iterations, the major concern is how to select the right time to reproduce, mutate and crossover. Furthermore, which allele(s) need to evolve. During this Phase I program, POC investigated the matrix manipulation of the GA algorithm. In order to maximize the GA algorithm, POC also investigated the adaptability of fuzzy logic to this approach. Since each iteration requires three evolution methods, it was necessary to determine which method is the most effective for a particular iteration.

Fuzzy logic is based on the uncertainty of the fuzzy set theory[18-19]. The fuzzy set theory works with the quantification of the meanings of words in graphs within the framework of set theory. Fuzzy theory is a mathematical theory, and what is called fuzziness takes in one aspect of uncertainty. Fuzziness is the ambiguity that can be found in the definition of a concept or the meaning of a word. For example, if we say "tall person", we cannot clearly determine who is tall and who is not. If we take a look at the ambiguity of the meaning of "tall" in terms of the expression of amount with a range of height of 5′ to 6.5′, the degree to which height x can be called "tall" is μ; that is, we make the height x correspond to degree μ, as shown in Figure 3-19.

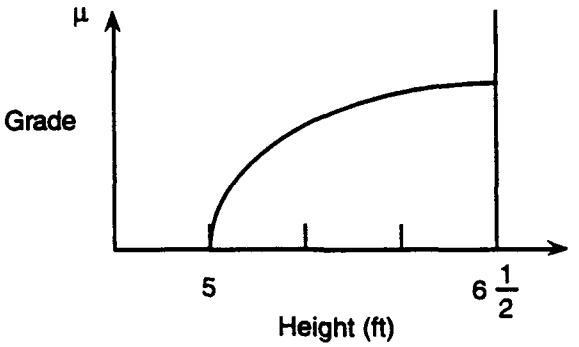

Figure 3-17.
Grade of Height.

Therefore, we can obtain many outputs; $0 \leq \mu \leq 1$. In conventional logic, the output is either "0" or "1". The main concern of POC's fuzzy logic adaptability to GA was how to utilize these many outputs. Since the GA is based on the iteration of a fitness comparison and an evolution operation, we used this fuzzy theory to reduce the number of iterations. In our investigations, we found that fuzzy rules can make decisions regarding which evolution step is more efficient to the particular iteration. For example, first n maximum fitness chromosomes are very close to each other; thus, it is better to perform mutation than crossover. On the other hand, if two high fitness alleles are not similar to each other, crossover provides a faster optimum search than mutation. In Table 3-1, the fuzzy rules for each necessary evolutionary step are shown. These rules are determined by the two fuzzy logic parameters Pi and qi, where Pi and qi are defined as follows:

$$p_i = \frac{\text{Fitness of } i^{th} \text{ Parent - Fitness of } (i-1)^{th} \text{ Parent}}{\text{Fitness of } i^{th} \text{ Parent}}$$

$$q_i = \frac{\text{Number of identical bits between two parents}}{\text{Number of bits of chromosome}}$$

Table 3-1. Fuzzy Rule to Control Genetic Operations

q_i \ p_i	$0 \leq p_i \leq$ threshold	threshold $\leq p_1 \leq 1$
$0 \leq q_i \leq$ threshold	Reproduction 100% Crossover	Reproduction 80% Crossover 20% Mutation
threshold $\leq q_i \leq 1$	Reproduction 50% Crossover 50% Mutation	Reproduction 100% Mutation

Fuzzy logic control has been successfully used in an increasing number of application areas, including task scheduling, robot arm manipulation, cement kiln control, railroad control, and pipeline control. These rule-based systems incorporate fuzzy linguistic variables into their rule set in order to model a human's "rule-of-thumb" approach to problem solving. As an example, "If two maximum chromosomes are very much similar based on a bit-wise comparison, perform more mutations than crossover".

GAs are powerful search algorithms based on the mechanics of natural genetics. They ensure the proliferation of equal solutions while investigating new solutions via a systematic information exchange utilizing probabilistic decisions. This probabilistic decision performs three basic evolution operations; reproduction, mutation, and crossover.

The main question here is: "What is the best evolution operation for each interaction?" This basic question leads to three specific questions:

1. How many chromosomes from the previous population need to be reproduced?
2. Which locus (string position) needs to be mutated?
3. What is the best crossover membership?

These questions can be answered using fuzzy logic control. Figure 3-18 shows a schematic of the basic design of an adaptive genetic algorithm based on a fuzzy logic controller.

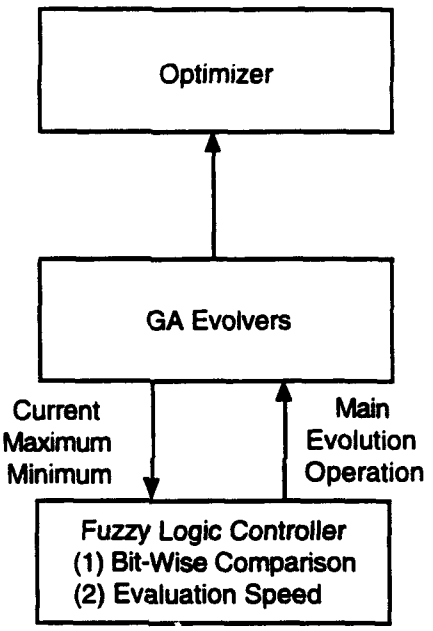

Figure 3-18.
Schematic Diagram of Fuzzy Logic Controlled GA Processor.

Reproduction is simply a process by which strings with large fitness values (good solutions to the problem at hand) receive correspondingly large numbers of copies in the new population. In tournament selection, pairs of strings compete with each other on a head-to-head basis for the right to be reproduced in the next generation. The participants in these competitions are selected based on the relative fitness of the strings, manifesting a "survival-of-the-fittest" atmosphere. The fitness values of two strings that are adjacent in the population (adjacent positions which are determined randomly) are compared. The string with the "best" fitness value is then selected. An advantage of tournament selection is that "best" can be defined as "highest" for a maximization problem, or as "lowest" for a minimization problem. Thus, tournament selection yields a linear rank-based selection. Actually, the particulars of the reproduction scheme are not critical to the performance of the GA; virtually any reproduction scheme that biases the population toward the fitter strings works well. Once the strings are reproduced, or copied for possible use in the next generation, they are placed in a mating pool where they await the action of the other two operators.

Strings exchange information via probabilistic decisions by the second operator, called crossover. Crossover provides a mechanism for strings to mix and match their desirable qualities through a random process. After reproduction, simple crossover proceeds in three steps. First, two newly reproduced strings are selected from the mating pool. Second, a uniform position along the two strings is selected at random. This is illustrated below, where two binary coded strings, A and B, of length 16 are shown aligned for crossover:

$$A=11111111/1111111$$

$$B=000000000/0000000.$$

Notice how crossing site 9 has been selected in this particular example through random choice, although any of the other 14 positions were just as likely to have been selected. The third step is to exchange all characters following the crossing site. The two new strings following this crossing are shown below as A' and B':

$$A'=11111111/0000000$$

$$B'=000000000/1111111.$$

String A' is made up of the first part of string A and the tail of string B. Likewise, string B' is made up of the first part of string B and the tail of string A. Although crossover uses random choice, it should not be thought of as a random walk through the search space. When combined with reproduction, it is an effective means of exchanging information and combining portions of high-quality solutions.

Reproduction and crossover give GA's the majority of their search power. The third operator, mutation, enhances the ability of the GA to find near-optimal solutions. Mutation is the occasional alteration of a value at a particular string position. It is an insurance policy against the permanent loss of any simple bit. A generation may be created that is void of a particular character at a given string position. For example, a generation may exist that does not have a 1 in the third string position when, due to the chosen coding, a 1 at the third position may be critical to obtaining a quality solution. Under these conditions, neither reproduction nor crossover will ever produce a 1 in this third position in subsequent generations.

In this Phase I program, POC showed the feasibility of applying fuzzy logic to GAs. The anticipated benefits from the application of fuzzy logic to GAs are:

(1) Efficient usage of parallel processor,
(2) Entire or partial elimination of redundant evolving procedure,
(3) Reduction of system complexity and architecture,
(4) Reduction of the required interconnectivity,
(5) Increase in overall processing speed, and
(6) An adaptive learning and processing capability.

3.4 Design of a Parallel Genetic Evolution System

The terms GA and neural network refer to classes of procedures and models that are developed as abstractions of information processing systems in nature. One of the reasons for these common biological foundations is that the natural evolutionary process has highly optimized computational engines. Such an optimized computational capability can be obtained through a parallel processing computing system. Since each GA system operates

on a population of individuals, a parallel computing approach is of great importance in speeding up the algorithm's execution. An initial design of a processing platform, including a multiple instruction/multiple data (MIMD) architecture based on a digital signal processor (DSP) computing environment, has been completed in this Phase I program. The flow chart for a parallel electronic genetic evolution system (PEGES) on a parallel processor architecture is shown in Figure 3-19. This process can be greatly accelerated through the use of a multiprocessor structure with high interconnectivity.

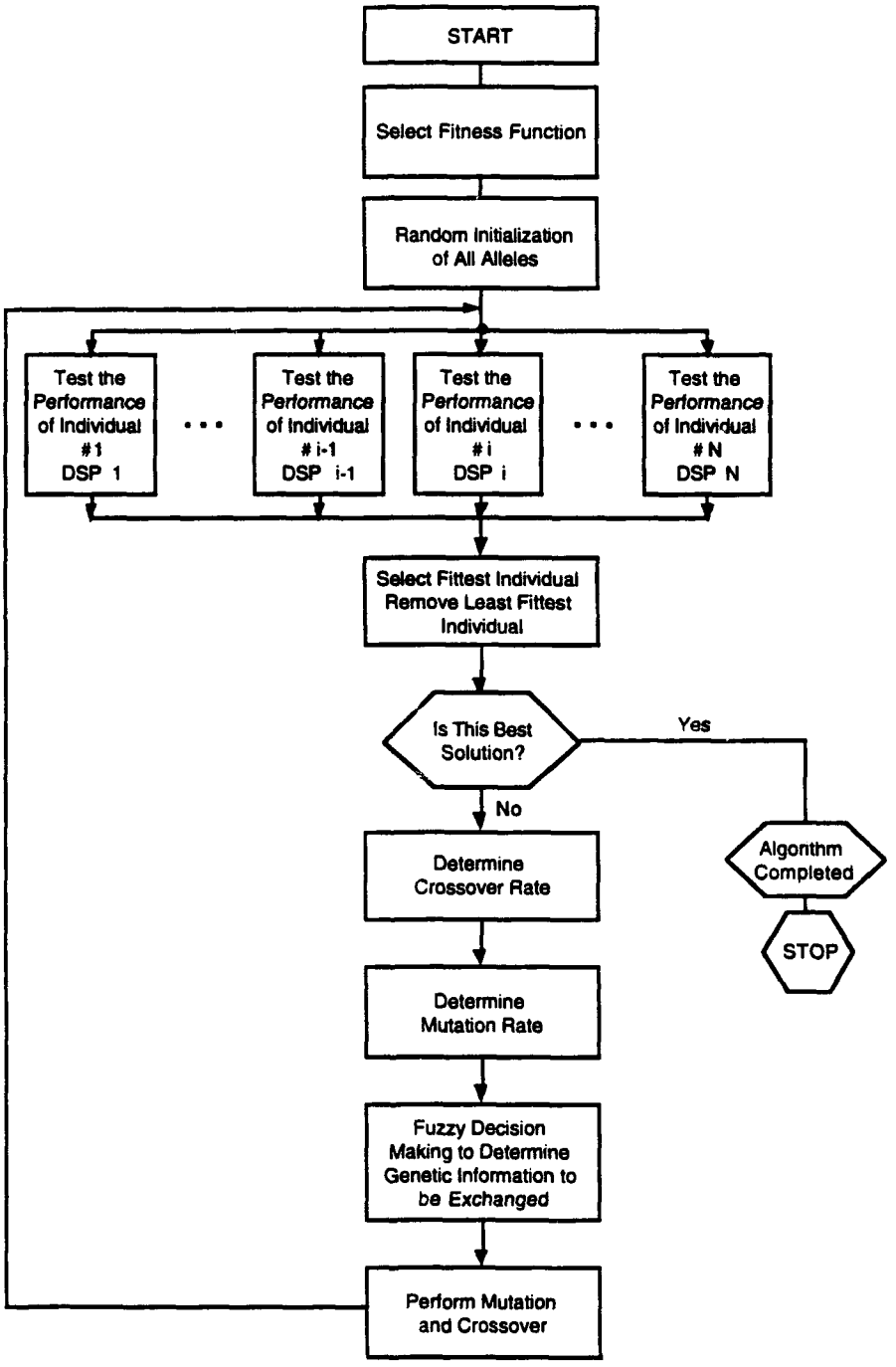

Figure 3-19.
Flow Chart of the PEGES.

3.4.1 Initial Design of DSP Parallel Processor

POC has good experience in the design of electronic multiprocessor systems. Based on this in-house experience, POC will finalize the Phase II prototype design in the early stages of the Phase II program. In this section, a description of a multiprocessor system previously developed at POC is presented. In the Phase II project, POC will increase the number of DSP processors and develop a more flexible optoelectronic interconnection network.

The processing power of POC's multiprocessor system is based on the Texas Instruments DSP chip (TMS320C40). Three processing layers (PLs) containing processing elements (PEs) are proposed in a 3-D configuration. Each layer contains 4 PEs, and each PE can communicate by a bi-directional link with a host computer. The communication between the host computer and the PEs uses a direct memory access (DMA) mode for fast data transfer. Each PE can access its local memory and a part of the global memory. The host computer can access the global memory only. The data exchange between the host computer and the PEs is obtained by updating the global memory addressable by the host computer. A conceptual graph showing the data transfer between the host computer and the PEs is depicted in Figure 3-20. The data transfer from the host computer to the PEs can be performed sequentially if each processor needs to receive a different set of data or can be broadcast in parallel if an identical data set is to be processed by each PE.

The developed multiprocessor system contains four DSP processors per layer. A total of three layers have been fabricated. Figure 3-21 shows a three-layer parallel multiprocessor. Each layer consists of an array of nodes. Each node contain one PE and one optical interconnect (OI) module. The optical interconnects are designed to be unidirectional for feed-forward data transfer. Figure 3-22 illustrates the design of each layer. Two communication schemes are used. A parallel fully interconnected network for four PEs (per layer) is employed in the node array to efficiently handle 2-D transfer of data to/from the host computer. This system bus solves the data transfer bottleneck between the host computer and the multi-processor system Due to this system bus design, the node in the multiprocessor system will not be interrupted and can continue to process data and simultaneously transfer data to/from the host computer. An additional RAM in each node will make this uninterrupted data transfer possible. The second interconnection system is used to provide vertical point-to-point communication between two layers of PEs. One of our main goals in this Phase II program is to improve the existing optical inter-layer communication (i.e., vertical communication) by employing a faster and reconfigurable many-to many interconnection scheme.

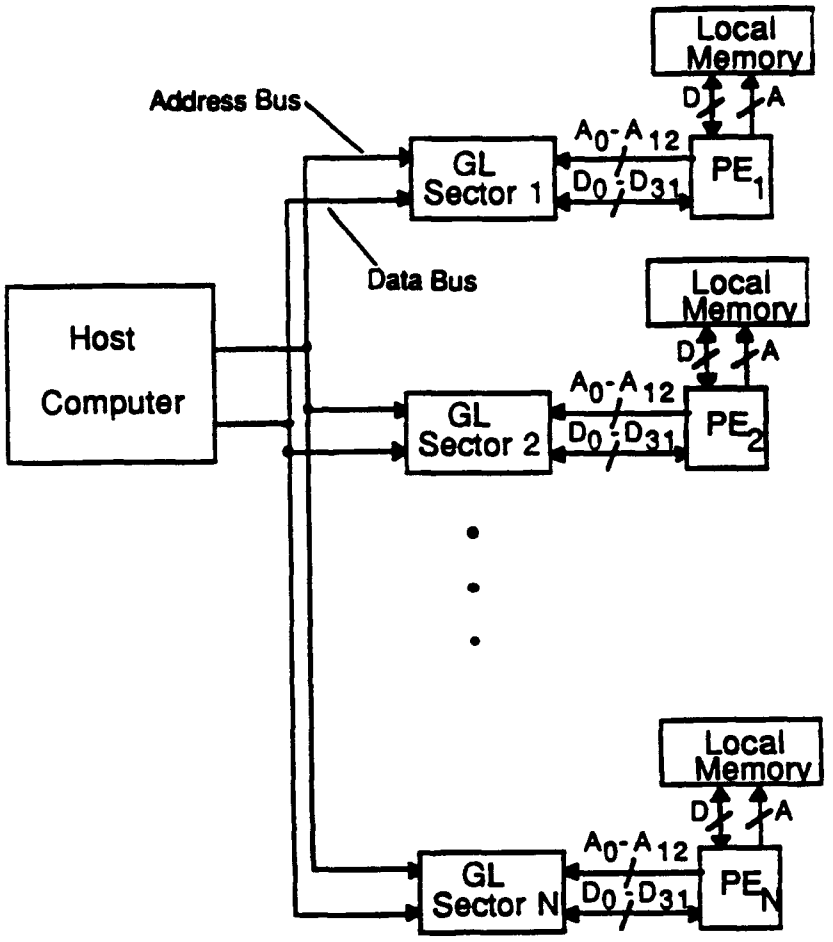

Figure 3-20.
Conceptual graph of the data transfer between the HC and PEs.

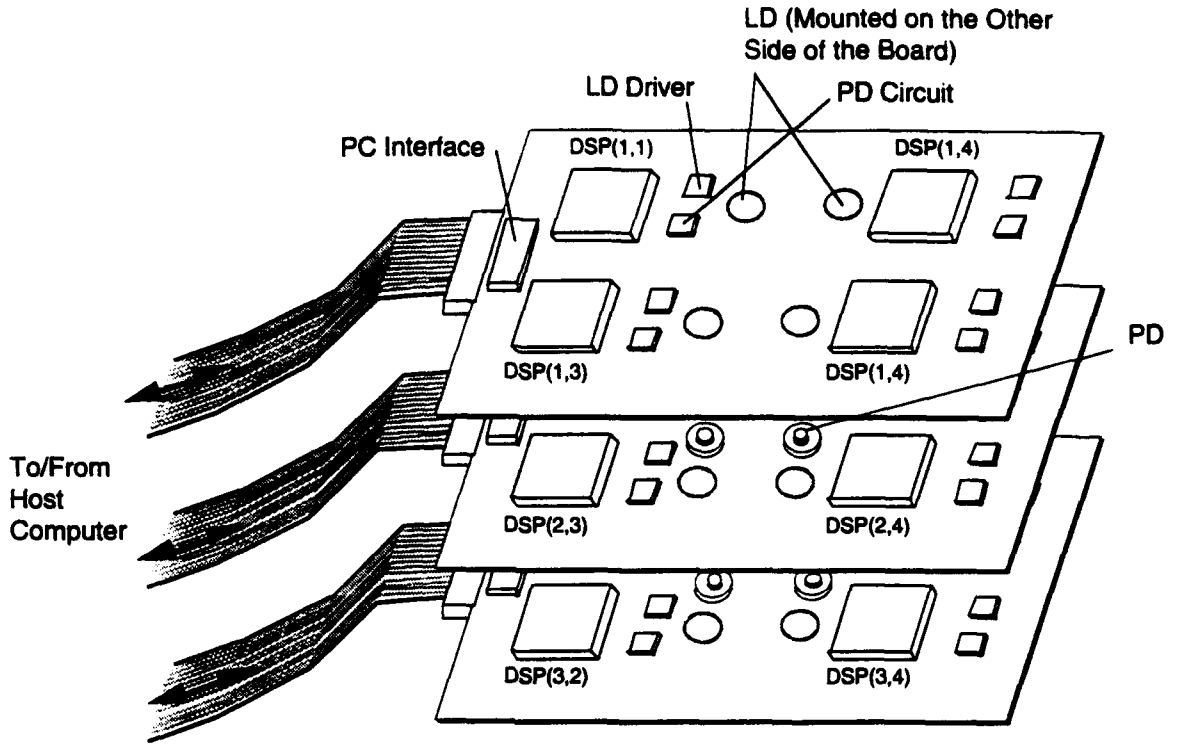

Figure 3-21.
Three layer multiprocessor system.

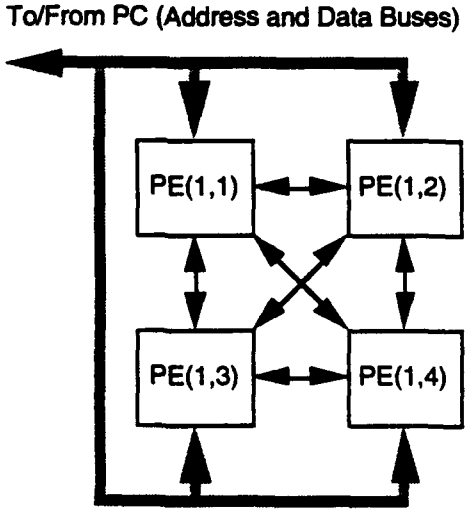

Figure 3-22.
Interconnection scheme for a single layer.

3.4.2 Mapping GA to Parallel Processor

Assume that we want to process one frame of an image as a chromosome. In this case, it is essential to achieve high interconnectivity as well as a high processing speed. Here, it is impossible to process such a large number of strings using a single processor. On the other hand, a multiprocessor with high interconnectivity (as described in the previous section) can be applied to image understanding issues. Unfortunately, there is one drawback to this, which is the mapping of the GA to parallel processing. This is available with the 3L parallel C programming language, which implements the same parallel processing model. Parallel C implements the modules of a block diagram as independent tasks that communicate via unidirectional channels.

Parallel C is a software environment for programming in multi-processor systems. It is specifically designed to implement the modular architecture of parallel networks, such as the DSP block diagram method of specifying algorithms. It is based on a model from parallel processing theory known as Communicating Sequential Processes (CSP). In this model, a computing system is a collection of concurrently active processes (or tasks), which can only communicate with each other over channels, as in Figure 3-23. A channel connects exactly one processor to exactly one other processor. Channels are unidirectional; two are required for communication in both directions. Each processor can have any number of channels. Channels are automatically synchronized; a sending processor must wait until the receiving processor is ready. Here a processor is treated as a "black box" connected to the outside world only by its channels. What is inside is not important; it could be another complex system of software or hardware.

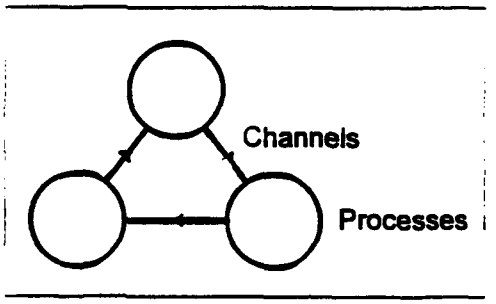

Figure 3-23.
CSP Model.

With Parallel C, an application comprises a number of these sequential tasks which communicate by means of unidirectional channels. These tasks are programs that execute standard C code, assembler, or DSP library function calls, and have their own region of memory for code and data. Each task has vectors of *input ports* and *output ports*; tasks can communicate with each other by arranging for the output port of one task to be connected to the input port of another task. Tasks can be treated as building blocks for parallel systems, to be wired together rather like electronic components. In addition to the static tasks, there is also the facility to create light-weight tasks called *threads* dynamically at run-time.

A *configurer* controls the placement of tasks and channels onto the available processor network. Two configuration tools are available. The *General Configurer* takes

information about both the logical and physical topology of the DSP network and combines multiple tasks into an application image suitable for execution. Assignment of tasks to processors is determined by a user-written text file called a configuration file. Changing the configuration only required re-running the configurer; recompilation and re-linking are not necessary. In particular, a multi-tasking application can be developed on a single processor and then adapted for a multi-processor network simply by changing the configuration file. The *Flood-Fill Configurer* allows creation of applications which run on any network of processors. At load time, the software analyzes the DSP network and automatically loads all processors with copies of a "worker" task. It then routes messages between the workers and a "master" task on the root DSP. Flood-Fill applications are therefore *topology-independent*. The communication between the master and the workers is taken care of automatically by the system software.

System debugging is of particular importance for Parallel DSP applications. With 3L Parallel C, debugging support is available for the analysis of multi-tasking and multi-processor systems. There is also a global I/O facility giving tasks on any processor the ability to write information to the screen of a host computer.

Signal processing theory, just like other electronic engineering disciplines, has made extensive use of block diagrams to describe algorithms and systems. These diagrams usually show complex networks of components all interconnected together. A signal processing block diagram like the one in Figure 3-24 describes a number of independent modules operating in parallel, just as in the DSP model. With Parallel C this can be directly implemented on a physical DSP network. Parallel C programming matches the structure of DSP algorithms. Because Parallel C is based a on a standard ANSI C and does not use special language extensions, programmers can work in an environment that they are used to and can make use of the large body of in-house, commercial, and public domain C code for DSP algorithms.

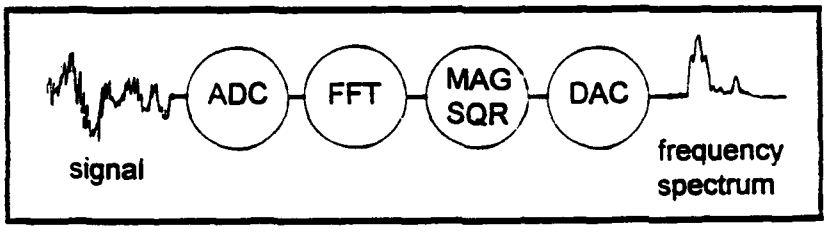

Figure 3-24.
Diagram of Independent Module Operating in Parallel.

4.0 POTENTIAL POST APPLICATIONS

The successful completion of the proposed nonlinear adaptive generic algorithm will lead to several near term applications for both the military and commercial sectors. Up to now, the construction of complex nonadaptive systems has resulted in systems where learning is difficult. The more evolutionary approach of genetic algorithm-based learning will provide extensible systems that will allow increasingly powerful learning and complexity. POC's nonlinear adaptive GA will provide the foundations of generic search and optimization, which can be applied to the following applications:

(1) Military applications
- •Multisensor data fusion
- •Vision sensing systems for unmanned vehicles
- •Threat assessment and decision module
- •Intelligent analysis
- •Automatic target recognition
- •Communication routing

(2) Commercial applications
- •Traveling salesman problem
- •VLSI circuit layout optimization
- •Communication network link size optimization and optimal routing
- •Optimization of multiprocessor routing
- •Image registration
- •Image feature searching
- •Highly sophisticated security and warning system
- •Pattern recognition
- •Intelligent manufacturing
- •Heuristic fault diagnosis
- •School bus routing
- •Neural network training
- •Adaptive document clustering
- •Recursive adaptive filter design
- •Holographic/diffractive optical element design

5.0 CONCLUSIONS AND RECOMMENDATIONS

In this Phase I program, POC successfully demonstrated a universal high dimensionality decision making GA, as well as provided an initial Phase I prototype design. The major achievements of this program include the determination of fuzzy logic adaptability to GA, the reevaluation of the mutation, the design of the GA evolvers, and a demonstration of the feasibility of the proposed GA to high dimensionality decision making.

In the Phase II program, POC will finalize the GA DSP parallel processor architecture, and will focus on the following tasks:

1. Finalization of the parallel GA processor.
2. Implementation of the Phase II prototype.
3. Development of the parallel programming.
4. Mapping the current GA program into the parallel programming.
5. Demonstration of image understanding using the Phase II prototype GA decision maker.

6.0 REFERENCES

1. E. Waltz and J. Llinas, *Multisensor Data Fusion*, Artech House, Boston (1990).
2. S.S. Iyengar, R.L. Kashyap, and R.N. Madan, Eds., "Special section on distributed sensor networks," *IEEE Trans. Sys. Man Cybernet.* **21**(5), 1027-1230 (1991).
3. White, F., *et al.*, "A Model for Data Fusion," SPIE Conference on Sensor Fusion, Orlando, FL. April 1988.

4. B.V. Dasarathy, "Paradigims for information processing in multisensor environments," in *Sensor Fusion III, Proc. SPIE* **1306,** 69-80 (1990).
5. D.W. Ruck, S.K. Rogers, M. Kabrisky, and J.P. Miller, "Multisensor target detection and classification," in *Sensor Fusion, Porc. SPIE* **931,** 14-21 (1988).
6. J.H. Holland, Adaptation in neural and artificial systems, Ann Arbor: The university of Michigan Press (1975).
7. K.T. Pal, "Genetic Algorithms for the traveling salesman problem based on a henistic crossover operation", Biol, Cybern, Vol. 69, No. 5-6, 539 (1993).
8. W.K. Lai and G.G. Coghill, "Genetic Breeding of Control Parameters for the Hopfield/Tank Neural Net", Proceedings of the IEEE International Conference on Neural Networks, Vol. IV, 141 (1992).
9. D. Dasgupta and D.R. McGregor, "Digital Image Registration using structured Genetic Algorithms", SPIE proceeding Vol. 1466, 226 (1992).
10. G. Seetharaman, O.S. Prabhu and A. Narasimhan, :"Two modified crossover and mutation operators for Image segmentation by Genetic Algorithms", SPIE Proceeding Vol. 1766, 66 (1992).
11. J. Gonzalez-Seco, "A Genetic Algorithm as the learning procedure for neural networks", Proceeding of the IEEE International Conference on Neural Networks, Vol. I 835 (1992).
12. J.G. Elias and B. Chang, "A Genetic Algorithm for Training Networks with Artificial Dendritic Trees", Proceedings of the IEEE International Conference on Neural Networks, Vol. I, 652 (1992).
13. A.V. Scherf and L.D. Voelz, "Training Neural Networks with Genetic Algorithms for Target Detection", SPIE proceeding Vol. 1910, 734 (1992).
14. M. Freidman, U. Mahlab and J. Shamir, "Collective Genetic Algorithm for Optimzation and its Electro-Optic Implementation", Appl. Opt. Vol. 32, 4423 (1993).
15. J. Cui, T.C. Fogarty and J.G. Gammack, "Searching Data Using Parallel Genetic Algorithms on a Transputer Computing", Future Generation Computer Systems, Vol. 9, 33 (1993).
16. A.R. Simpson and S.D. Priest, "The Application of Genetic Algorithms to Optimization Problems in Geotechnics", Comp. Geotech. Vol. 15, 1 (1993).
17. E.S. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling", IEEE Trans. Parallel and Distributed Syst. Vol. S. 113 (1994).
18. L.A. Zadeh, "Fuzzy Sets", Information and control, Vol. 8, 338, (1965).
19. H. Zimmerman, Fuzzy Set Theory-and its Application, 2nd Ed. Boston: Kluwer, (1990).

APPENDIX

```
/*****************************************************************
    The program with choosing of function input
*****************************************************************/


#include<Float.h>
#include<graph.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>
#include<malloc.h>
#include<string.h>
#include<fcntl.h>


int n,i,j,t,k,tt,kk,pp,ngene,slength,geneval,nbit[11], rep = 0, cross;
int p,int_rand,itt_rand,intm_pxval,it2_rand,intn,nb,nbold,ne,neold;
double fitness[6000],maximum2,minimum2,maximum,minimum, mold, minold,mold2,minold2;
double gendec, chrof[6000][20], diffmax, diffmin,fac;
int cbit[200], beg[200], dbit;
int maxparent[200],minparent[200],maxparent2[200],minparent2[200];
int quot,rem,modd,func,imaxt,imint,imax2,imin2,roop;
static int huge bit_val[6000][200];
char string[10];
long mod[10];
int percent, kol;
double a, b, c, d, e, f, g, h, v, m, o, l, res;
int choice, res_bit;
int nbrep, nerep, nbmut, nemut, nbmutval, nemutval, nbmutmin, nemutmin,
    nbmutmv, nemutmv, nbdlmut2, nedlmut2, nbdlmut, nedlmut, nbcross, necross;

/*struct videoconfig config;*/
short wx, wy, xb, yb, lw, hw;


ldiv_t result;/*        IMPORTANT CHANGE !!!            */
```

```
div_t result2;
div_t result3;

FILE *fi;

void comparison(void);
void fitfun(void);
void refitfun(void);
void max1(void);
void min1(void);
void remax1(void);
void remin(void);
void intpat(void);
void reproduction(void);
void mutation(void);
void mutval(void);
void mutmin(void);
void mutminval(void);
void crossover(void);
void decigene(void);
void max_min (double* ord);
void dlmut(void);
void dlmut_twice(void);
void crossover1(void);



double f1(double x, double y);
double f2(double x, double y);
double f3(double x, double y, double p, double q);
double f4( double x1, double x2, double x3, double x4, double x5,
        double x6, double x7, double x8, double x9, double x10);
double f5( double x1, double x2);


void main(void)

{

    fi = fopen ("res.dat", "w");
```

```
printf("WHAT FUNCTION DO YOU WANT TO CHOOSE?\n");
printf("\n(1) F1(x,y) = a*(x^b - d)*(y^c - f) + g;\n");
printf("\n(2) F2(x,y) =  (x-2^c)^d + (y-2^g)^h - 2^o\n");
printf("\n(3) F3(x1,x2,x3,x4) = a*x1*x2*x3 - b*x2*x3*x4\n");
printf("\n(4) F4(x1,...,x10) = x1*x2 - x3*x4 + x5*x6 - x7*x8 + x9*x10 \n");
printf("                              - x1*x5*x9\n");
printf("\n(5) Banana function F5(x1, x2) = 562500. - 22500. * x1 + 225.* x1*x1 + \n");
printf(" 9.* x1*x1*x1*x1 - 750. * x1*x1 * x2 + 15625. * x2*x2; \n");
printf("\nTYPE 1, 2, 3, 4 or 5\n");
scanf("%d", &choice);
printf("choice = %d\n" , choice);


switch(choice)
{
    case(1):
        ngene = 2;
        printf("\n(1) F1(x,y) = a*(x^b - d)*(y^c - f) + g;\n");
        printf("\nInput a  b  d  c  f  g\n");
        scanf("%lf %lf %lf %lf %lf %lf", &a, &b, &d, &c, &f, &g);
        break;
    case(2):
        ngene = 2;
        printf("\n(2) F2(x,y) = (x-2^c)^d + (y-2^g)^h - 2^o\n");
        printf("\nInput  c  d  g  h  o \n");
        scanf("%lf %lf %lf %lf %lf",
                 &c, &d, &g, &h, &o);
        fprintf(fi, "%.11f %.11f %.11f %.11f %.11f\n",
                     c, d, g, h, o);
        break;
    case(3):
        ngene = 4;
        printf("\n(3) F3(x,y,p,q) = a*x*y*p + b*y*p*q\n");
        printf("\nInput a and b\n");
        scanf("%lf %lf", &a, &b);
        break;
    case(4):
        ngene = 10;
        break;
    case(5):
```

```
            ngene = 2;
            break;
        default:
            break;
    }



       /* This part is to determine the parameters such as the number of genes,
       the number of bit of each gene and genechrom. */

/*  printf("How many genes? \n");
    scanf("%d", &ngene);*/

    nbit[0]=0;

/* printf("type the factor? \n");
    scanf("%lf", &fac);*/

    fac = 1.5;




/*  printf("How many bits for crossover? \n");
    scanf("%d", &cross);                    */

    cross = 5;
    printf("How many bits for resolution? \n");
    scanf("%d", &res_bit);
    res = pow (2., (double)res_bit);
    /*_clearscreen(_GCLEARSCREEN);*/

    for (i=1; i<=ngene; i++)
        {
         printf("How many bit for %d gene? \n", i);
         scanf("%d", &nbit[i]);
         nbit[i] = nbit[i] + res_bit;
```

```
    }


    slength=0;
    printf("The number of genes is %d. \n", ngene);
/*  fprintf(fi,"The number of genes is %d. \n", ngene);*/

    for (i=1; i<=ngene; i++)
        {
        /* printf("The %d gene is a %d-bit string. \n", i, nbit[i]); */
        /*fprintf(fi, "The %d gene is a %d-bit string. \n", i, nbit[i]);*/
           slength=slength+nbit[i];
        }
    /* printf("The total number of bit for each genechrom is %d\n",slength);*/
    /*  fprintf(fi,"The total number of bit for each genechrom is %d\n",slength);*/
    /* This part is to calculate the modular of each gene */

    for (j=1; j<=ngene; j++)
        {
         mod[j]=1;
         for (i=1; i<=nbit[j]; i++)
         {
          mod[j]=mod[j]*2;
          /*printf("The mod[j]=%d\n",mod[j]); */
  /*           fprintf(fi,"The mod[j]=%ld\n",mod[j]);*/
         }
         kk=1;
/*         printf("The modular of %d gene is %d \n", j,mod[j]);
           fprintf(fi,"The modular of %d gene is %d \n", j,mod[j]); */
        }

    /* This part of program is to initialize the all the matrix elements
       with logic value "0". */

         it2_rand=rand();
         result3=div(it2_rand,14);
         intn=result3.rem+10;
    intn=100;

    for (i=1; i<=intn; i++)
```

```
        {
        for (j=1; j<=slength; j++)
        {
         bit_val[i][j]=0;
         /* printf("The matrix[%d][%d] is %d \n", i,j,bit_val[i][j]);
            fprintf(fi, "The matrix[%d][%d] is %d \n", i,j,bit_val[i][j]); */
        }
        }

    /* This part of program is to generate a number (8) of randomiy selected
       initial genes */

    for (i=1; i<=intn; i++)
        {
        for (j=1; j<=ngene; j++)
        {
         chrof[i][j]=0.;
           /* printf("The random number:%d, modular:%d \n", chr_rand, chrof[i][j]); */
           /* fprintf(fi, "The random number:%d, modular:%d \n", chr_rand, chrof[i][j]); */
        }
        }

    for (i=1; i<=intn; i++)
        {
        for (j=1; j<=ngene; j++)
        {
         int_rand=rand();
         result=ldiv((long)int_rand,mod[j]);
         chrof[i][j]=(double)result.rem;
         /* fprintf(fi, "chrof[%d][%d]=%f \n", i,j,chrof[i][j]);

                    printf("chrof[%d][%d]=%f \n", i,j,chrof[i][j]);    */
        }
        }

/* This part of program is to evaluate the value of all the matrix value */

    intm_pxval=0;
/*  fprintf(fi, "\n");*/

    for (i=1; i<=intn; i++)
```

```
        {
          intm_pxval=0;
          for (j=1; j<=ngene; j++)
          {
           geneval=(int)chrof[i][j];
           for (t=1; t<=nbit[j]; t++)
               {
            k=intm_pxval + t;
            result=ldiv((long)geneval,(long)2);
            bit_val[i][k]=(int)result.rem;
            geneval=(int)result.quot;
              /*printf("bit_val[%d][%d]=%d\n",i,k,bit_val[i][k]);*/
              /*fprintf(fi, "bit_val[%d][%d]=%d\n",i,k,bit_val[i][k]);*/
            if (t==nbit[j])
               {
                intm_pxval=intm_pxval+nbit[j];
               }
           }
          }
        }


/*  for (i=1; i<=intn; i++)
        {
         for (j=1; j<=slength; j++)
         {
          printf("%d",bit_val[i][j]);
          fprintf(fi, "%d",bit_val[i][j]);
          if (j==slength) fprintf(fi, "\n");
         }
        }
        for(i=1; i<=intn; i++){
            fprintf(fi, "i = %d ", i);
            for (j = 1; j <= slength; j++){
                fprintf(fi, "%d",bit_val[i][j]);
                if (j==slength) fprintf(fi, "  fitness=%lf\n", fitness[i]);
            }
        }   */


/*  fprintf(fi, " ********************************************************* \n");
```

```
    fprintf(fi, " This is the initial random generation result. \n");
    fprintf(fi, " The chromesomes are \n");*/

    fitfun();

/*  for(i=1; i<=intn; i++){
            fprintf(fi, "i = %d ", i);
            for (j = 1; j <= slength; j++){
                fprintf(fi, "%d",bit_val[i][j]);
                if (j==slength) fprintf(fi, "  fitness=%lf\n", fitness[i]);
            }
        }*/

    max1();
    min1();
    intpat();
    mold = maximum;
    minold = minimum;
    mold2 = maximum2;
    minold2 = minimum2;

   /*   for (i=1; i<=intn; i++){
        fprintf(fi, "i = %d ", i);
        for (j = 1; j <= slength; j++)
        {
        fprintf(fi, "%d", bit_val[i][j]);
        if (j==slength) fprintf(fi, "\n");
        }
        }        */

for (roop=1; roop<=100; roop++){

    printf("roop = %d\n", roop);
    fprintf(fi, "roop = %d\n", roop);
    fprintf(fi, " ********************************************************** \n");
    fprintf(fi, " The %d iteration result is following: \n", roop);
/*  fprintf(fi, " The chromesomes are \n");*/
    nb=0;
    ne=0;
    reproduction();
    comparison();
```

```
    /*printf("\ndbit = %d\n", dbit);
    printf("maximum = %lf mold = %lf \n", maximum, maximum2);
    diffmax = (maximum - maximum2) / maximum;
    diffmin = (minimum - minimum2) / minimum;*/
    if (dbit <= (int)(slength/fac)/* && ((diffmax >= 0.1) || (diffmin >= 0.1))*/){
        crossover();
        crossover1();
        decigene();
        refitfun();
        remax1();
        remin();
        intpat();
        printf("No mutation!  \n");
    }
    else{
        mutation();
        mutval();
        mutmin();
        mutminval();
        if ( choice == 5 || choice == 2)
            dlmut();
    /*  dlmut_twice();
        crossover();    */
        decigene();
        refitfun();
        remax1();
        remin();
        intpat();
        printf("No crossover!  \n");
    }
    fprintf(fi, "The maximum fitness is the %d chromesome, the value is %e. \n", imaxt, maximum);
    fprintf(fi, "The minimum fitness is the %d chromesome, the value is %e. \n", imint, minimum);
    fprintf(fi, "The second maximum fitness is the %d chromesome, the value is %e. \n", imax2, maximum2);
    fprintf(fi, "The second minimum fitness is the %d chromesome, the value is %e. \n", imin2, minimum2);

/*  fprintf(fi,"\nimint = %d imaxt =%d\n", imint, imaxt);
    fprintf(fi, "\nmax = %e x = %e, y = %e\n", maximum, chrof[imaxt][1]/res, chrof[imaxt][2]/res);*/
/*  for (j=1; j<=slength; j++)
        fprintf(fi, "%d",bit_val[imaxt][j]);*/
    fprintf(fi,"\n");
/*  fprintf(fi, "\nmin = %e x = %e, y = %e\n", minimum, chrof[imint][1]/res, chrof[imint][2]/res);
```

```
    for (j=1; j<=slength; j++)
        fprintf(fi, "%d", bit_val[imint][j]);*/

/*  for(i=1; i<=ne; i++){

        if (i == nbrep) fprintf (fi, "----repetition----\n");
        if (i == nbmut) fprintf (fi, "----mutation----\n");
        if (i == nbmutval) fprintf (fi, "----mutval----\n");
        if (i == nbmutmin) fprintf (fi, "----mutmin----\n");
        if (i == nbmutmv) fprintf (fi, "----mutminval----\n");
        if (i == nbdlmut) fprintf (fi, "----dlmut----\n");
        if (i == nbcross) fprintf (fi, "----crossover----\n");
        if (i == nbdlmut2) fprintf (fi, "----dlmut_twice----\n");
        fprintf(fi, "i = %d ", i);
        for (j = 1; j <= slength; j++){
            fprintf(fi, "%d",bit_val[i][j]);
            if (j==slength) fprintf(fi, "  fitness=%e\n", fitness[i]);
        }
    }*/

      if ((maximum == mold) && (minimum == minold) &&
        (imaxt == 1) && (imint == 3) && (imax2 == 2) && (imin2 == 4)){
        printf("The end\n");
        printf("Iteration number %d\n", roop);
        printf("\nmin = %e x1=%.1e, x2=%.1e x3=%.1e x4=%.1e x5=%.1e x6=%.1e x7=%.1e x8=%.1e x9=%.1e x10=%.
1e\n",
        minimum, chrof[imint][1]/res, chrof[imint][2]/res, chrof[imint][3]/res,
        chrof[imint][4]/res, chrof[imint][5]/res, chrof[imint][6]/res, chrof[imint][7]/res,
        chrof[imint][8]/res, chrof[imint][9]/res, chrof[imint][10]/res);

/*      fprintf(fi, "\nmin = %e x1=%.1e, x2=%.1e x3=%.1e x4=%.1e x5=%.1e x6=%.1e x7=%.1e x8=%.1e x9=%.1e x
10=%.1e\n",
        minimum, chrof[imint][1]/res, chrof[imint][2]/res, chrof[imint][3]/res,
        chrof[imint][4]/res, chrof[imint][5]/res, chrof[imint][6]/res, chrof[imint][7]/res,
        chrof[imint][8]/res, chrof[imint][9]/res, chrof[imint][10]/res);

        fprintf(fi, "\nmax = %e x1=%.1e, x2=%.1e x3=%.1e x4=%.1e x5=%.1e x6=%.1e x7=%.1e x8=%.1e x9=%.1e x
10=%.1e\n",
        maximum, chrof[imaxt][1]/res, chrof[imaxt][2]/res, chrof[imaxt][3]/res,
        chrof[imaxt][4]/res, chrof[imaxt][5]/res, chrof[imaxt][6]/res, chrof[imaxt][7]/res,
        chrof[imaxt][8]/res, chrof[imaxt][9]/res, chrof[imaxt][10]/res);*/
```

```
        break;
      }

  /*fprintf(fi,"mold = %lf  maximum = %lf\n", mold, maximum);
    fprintf(fi,"minold = %lf  minimum = %lf\n", minold, minimum);*/

    mold = maximum;
    minold = minimum;
    mold2 = maximum2;
    minold2 = minimum2;
}
     fclose(fi);
}


void fitfun(void)
{
        switch(choice)
        {
        case(1):
            for (i=1; i<=intn; i++){
                fitness[i] = f1(chrof[i][1]/res, chrof[i][2]/res);
                fprintf(fi, "The fitness of %d chromesome is %f \n", i, fitness[i]);
            }
            break;
        case(2):
            for (i=1; i<=intn; i++){
                fitness[i] = f2(chrof[i][1]/res, chrof[i][2]/res);
            }
            break;
        case(3):
            for (i = 1; i <= intn; i++){
                fitness[i] = f3(chrof[i][1]/res, chrof[i][2]/res,
                        chrof[i][3]/res, chrof[i][4]/res);
                fprintf(fi, "The fitness of %d chromesome is %f \n", i, fitness[i]);
            }
            break;
        case(4):
            fprintf(fi, "\nfitfun\n");
            for (i = 1; i <= intn; i++){
                fitness[i] = f4(chrof[i][1]/res, chrof[i][2]/res, chrof[i][3]/res,
```

```
                chrof[i][4]/res, chrof[i][5]/res, chrof[i][6]/res, chrof[i][7]/res,
                chrof[i][8]/res, chrof[i][9]/res, chrof[i][10]/res);

                fprintf(fi, "\nfitness = %.11f x1=%.11f, x2=%.11f x3=%.11f x4=%.11f x5=%.11f x6=%.11f x7=%
.11f x8=%.11f x9=%.11f x10=%.11f\n",
                fitness[i], chrof[i][1]/res, chrof[i][2]/res, chrof[i][3]/res,
                chrof[i][4]/res, chrof[i][5]/res, chrof[i][6]/res, chrof[i][7]/res,
                chrof[i][8]/res, chrof[i][9]/res, chrof[i][10]/res);
            }
            break;
        case(5):
            for (i=1; i<=intn; i++){
                fitness[i] = f5(chrof[i][1]/res, chrof[i][2]/res);
            }
            break;
        default:
            ;
        }
}

void refitfun (void)
{
    fprintf(fi, "\n");
    switch(choice)
        {
        case(1):
            for (i=1; i<=ne; i++){
                fitness[i] = f1(chrof[i][1]/res, chrof[i][2]/res);
                fprintf(fi, "The fitness of %d chromesome is %f \n", i, fitness[i]);
            }
            break;
        case(2):
            for (i=1; i<=ne; i++){
                fitness[i] = f2(chrof[i][1]/res, chrof[i][2]/res);
            }
            break;
        case(3):
            for (i=1; i<=ne; i++){
                fitness[i] = f3(chrof[i][1]/res, chrof[i][2]/res,
                        chrof[i][3]/res, chrof[i][4]/res);
                fprintf(fi, "The fitness of %d chromesome is %f \n", i, fitness[i]);
```

```
            }
            break;
        case(4):
            fprintf(fi, "\nre-fitfun\n");
            for (i = 1; i <= ne; i++){

                fitness[i] = f4(chrof[i][1]/res, chrof[i][2]/res, chrof[i][3]/res,
                chrof[i][4]/res, chrof[i][5]/res, chrof[i][6]/res, chrof[i][7]/res,
                chrof[i][8]/res, chrof[i][9]/res, chrof[i][10]/res);

/*              fitness[i] = chrof[i][1]/res * chrof[i][2]/res - chrof[i][3]/res *
                chrof[i][4]/res + chrof[i][5]/res * chrof[i][6]/res - chrof[i][7]/res *
                chrof[i][8]/res - chrof[i][9]/res * chrof[i][10]/res;
                */
/*              fprintf(fi, "\nfitness = %.11f x1=%.11f, x2=%.11f x3=%.11f x4=%.11f x5=%.11f x6=%.11f x7=%
.11f x8=%.11f x9=%.11f x10=%.11f\n",
                fitness[i], chrof[i][1]/res, chrof[i][2]/res, chrof[i][3]/res,
                chrof[i][4]/res, chrof[i][5]/res, chrof[i][6]/res, chrof[i][7]/res,
                chrof[i][8]/res, chrof[i][9]/res, chrof[i][10]/res);*/

            }
            break;
        case(5):
            for (i = 1; i <= ne; i++)
                fitness[i] = f5(chrof[i][1]/res, chrof[i][2]/res);
            break;
        default:
            ;
        }
}


void max1(void)
{
    maximum=fitness[1];
    imaxt=1;
    for (i=1; i<=intn; i++)
        {
         if (fitness[i]>=maximum)
        {
```

```
            maximum=fitness[i];
            imaxt=i;
           }
            else
           {
            maximum=maximum;
           }
           }
       printf("The maximum is %lf\n", maximum);
    if (imaxt==1)
        {
         maximum2=fitness[2];
         imax2=2;
        }
    else
        {
         maximum2=fitness[1];
         imax2=1;
        }


    for (i=1; i<=intn; i++)
        {
         if (fitness[i]>maximum2 && fitness[i]!=maximum && i != imaxt)
        {
         maximum2=fitness[i];
         imax2=i;
        }
         else
        {
         maximum2=maximum2;
        }
        }
}


void remax1(void)
{
    imaxt = 1;
    maximum = fitness[1];
```

```
for (i=1; i<=ne; i++){
     if (fitness[i] > maximum)
    {
     maximum = fitness[i];
     imaxt = i;
    }
     else if (fitness[i] == maximum)
    {
     imaxt = imaxt;
     maximum=fitness[i];
    }
     else
    {
     imaxt=imaxt;
     maximum=maximum;
    }
}

if (imaxt==1)
    {
     maximum2=fitness[2];
     imax2=2;
    }
else
    {
     maximum2=fitness[1];
     imax2=1;
    }

for (i=1; i<=ne; i++)
    {
     if (fitness[i]>maximum2 && fitness[i]!=maximum)
    {
     maximum2=fitness[i];
     imax2=i;
    }
     else
    {
     maximum2=maximum2;
    }
    }
```

```
      /*   printf("The maximum finding is finished\n");*/
}


void min1(void)
{
    imint = 3;
    minimum = maximum;
    for (i=1; i<=intn; i++)
        {
         if (fitness[i]<=minimum)
        {
         minimum=fitness[i];
         imint=i;
        }
         else
        {
         imint=imint;
         minimum=minimum;
        }
        }

    if (imint==1)
       {
        minimum2=fitness[2];
        imin2=2;
       }
    else
       {
        minimum2=fitness[1];
        imin2=1;
       }

    for (i=1; i<=intn; i++)
        {
         if (fitness[i]<minimum2 && fitness[i]!=minimum)
        {
         minimum2=fitness[i];
         imin2=i;
        }
```

```
        else
        {
         imin2=imin2;
         minimum2=minimum2;
        }
        }


    fprintf(fi, "\nThe 1st maximum fitness is the %d chromesome, the value is %f. \n", imaxt, maximum);
    fprintf(fi, "\nThe 1st minimum fitness is the %d chromesome, the value is %f. \n", imint, minimum);
    fprintf(fi, "\nThe 1st second maximum fitness is the %d chromesome, the value is %f. \n", imax2, maxim
um2);
    fprintf(fi, "\nThe 1st second minimum fitness is the %d chromesome, the value is %f. \n", imin2, minim
um2);

}

void remin(void)
{
    imint=1;
    minimum = maximum;
    for (i=1; i<=ne; i++){

         if (fitness[i]<minimum){
             minimum=fitness[i];
             imint=i;
         }
         else if(fitness[i]==minimum){
             minimum=fitness[i];
             imint=imint;
         }
         else{
             minimum=minimum;
             imint=imint;
         }

    }

    if (imint==1){
        minimum2=fitness[2];
        imin2=2;
```

```
    }
    else{
        minimum2=fitness[1];
        imin2=1;
    }


    for (i=1; i<=ne; i++){
         if (fitness[i]<minimum2 && fitness[i]!=minimum){
            minimum2=fitness[i];
            imin2=i;
         }
         else{
            minimum2=minimum2;
            imin2 = imin2;
         }
    }

/*    printf("The maximum fitness is the %d chromesome, the value is %lf. \n", imaxt,maximum);
    printf("The minimum fitness is the %d chromesome, the value is %lf. \n", imint,minimum);
    printf("The second maximum fitness is the %d chromesome, the value is %lf. \n", imax2,maximum2);
    printf("The second minimum fitness is the %d chromesome, the value is %lf. \n", imin2,minimum2);
    fprintf(fi, "The maximum fitness is the %d chromesome, the value is %lf. \n", imaxt,maximum);
    fprintf(fi, "The minimum fitness is the %d chromesome, the value is %lf. \n", imint,minimum);
    fprintf(fi, "The second maximum fitness is the %d chromesome, the value is %lf. \n", imax2,maximum2);
    fprintf(fi, "The second minimum fitness is the %d chromesome, the value is %lf. \n", imin2,minimum2);*
/

}

void intpat(void)
{
    for (j=1; j<=slength; j++)
         {
          maxparent[j]=bit_val[imaxt][j];
          minparent[j]=bit_val[imint][j];
          maxparent2[j]=bit_val[imax2][j];
          minparent2[j]=bit_val[imin2][j];
         }
    /*     printf("The parent finding is finished\n");*/
}
```

```
void reproduction(void)
{
    /* This part of program is to perform the reproduction of the maximum
       fitness chromesome */
    nb=1;
    ne=4;
    neold=0;
    nbold=0;
    for (j=1; j<=slength; j++)
        {
         bit_val[1][j]=maxparent[j];
         bit_val[2][j]=maxparent2[j];
         bit_val[3][j]=minparent[j];
         bit_val[4][j]=minparent2[j];
         }
       /*   fprintf(fi, "-------reproduction--------\n");
    for (i = nb; i <= ne; i++)
        {
        fprintf(fi, "i = %d ", i);
        for (j = 1; j <= slength; j++)
        {
        fprintf(fi, "%d",bit_val[i][j]);
        if (j==slength) fprintf(fi, "\n");
        }
    }
         printf("The reproduction is finished\n");*/
       nbrep = nb;
       nerep = ne;
       nbold=nb;
       neold=ne;

}


void mutation(void)
{
    /* This part of program is to perform mutation of the maximum
       fitness chromesome */
```

```
nb=neold+1;
ne=neold+slength;

for (i=nb; i<=ne; i++)
    {
     for (j=1; j<=slength; j++)
     {
      if (i-neold != j)
         bit_val[i][j]=maxparent[j];
      else
         if (bit_val[1][j] == 0)
         bit_val[i][j]=1;
         else
         bit_val[i][j]=0;
     }
    }
nbmut = nb;

nbold=nb;
neold=ne;
nb=neold+1;
ne=neold+slength;

for (i=nb; i<=ne; i++)
    {
     for (j=1; j<=slength; j++)
     {
      if (i-neold != j)
         bit_val[i][j]=maxparent2[j];
      else
         if (bit_val[2][j] == 0)
         bit_val[i][j]=1;
         else
         bit_val[i][j]=0;
     }
    }
   /*   fprintf(fi, "--------mutation---------\n");
for (i=nbold; i<=ne; i++)
    {
    fprintf(fi, "i = %d ", i);
    for (j = 1; j <= slength; j++)
```

```
        {
        fprintf(fi, "%d",bit_val[i][j]);
        if (j==slength) fprintf(fi, "\n");
        }
        } */
    nbold=nb;
    neold=ne;
    nemut = ne;
}


void mutval(void)
{
    /* This part of program is to perform mutation of the maximum
       fitness chromesome */

    nb=neold+1;
    ne=neold+slength;

    for (i=nb; i<=ne; i++)
        {
         for (j=1; j<=slength; j++)
         {
          bit_val[i][j]=maxparent[j];
         }
        }

    for (i=nb; i<=ne; i++)
        {
         it2_rand=rand();
         result3=div(it2_rand,3);
         n=result3.rem+1;
         int_rand=rand();
         result=ldiv((long)int_rand,(long)(slength-n-1));
         p=(int)result.rem+1;

          for (j=i-neold; j<=i-neold+n; j++)
              {
               if (bit_val[i][j] == 0)
              bit_val[i][j]=1;
               else
```

```
            bit_val[i][j]=0;
            }
    }
nbmutval = nb;

nbold=nb;
neold=ne;
nb=neold+1;
ne=neold+slength;

for (i=nb; i<=ne; i++)
    {
     for (j=1; j<=slength; j++)
     {
      bit_val[i][j]=maxparent[j];
     }
    }
tt=0;
for (k=1; k<=ngene; k++)
    {
     tt=tt+nbit[k-1];
     for (i=nb+tt; i<=nb+tt+nbit[k]; i++)
     {
      for (j=tt+1; j<=i-neold; j++)
           {
            if (bit_val[i][j] == 0)
           {
            bit_val[i][j]=1;
           }
            else
           {
            bit_val[i][j]=0;
           }
           }
      }
     }
/*fprintf(fi, "---------mutval---------\n");
for (i=nb; i<=ne; i++){

    fprintf(fi, "i = %d ", i);
    for (j = 1; j <= slength; j++)
```

```
        {
        fprintf(fi, "%d", bit_val[i][j]);
        if (j==slength) fprintf(fi, "\n");
        }
        } */
    nbold=nb;
    neold=ne;
    nemutval = ne;
}


void mutmin(void)
{
    /* This part of program is to perform mutation of the maximum
       fitness chromesome */
    nb=neold+1;
    ne=neold+slength;
    nbmutmin = nb;

    for (i=nb; i<=ne; i++)
        {
         for (j=1; j<=slength; j++)
         {
          if (i-neold != j)
             bit_val[i][j]=minparent[j];
          else
             if (bit_val[3][j] == 0)
             bit_val[i][j]=1;
             else
             bit_val[i][j]=0;
         }
        }
    nbold=nb;
    neold=ne;
    nb=neold+1;
    ne=neold+slength;

    for (i=nb; i<=ne; i++)
        {
         for (j=1; j<=slength; j++)
         {
```

```
            if (i-neold != j)
               bit_val[i][j]=minparent2[j];
            else
               if (bit_val[4][j] == 0)
               bit_val[i][j]=1;
               else
               bit_val[i][j]=0;
          }
         }

    /*fprintf(fi, "----------mutmin---------\n");
    for (i=nb; i<=ne; i++)
         {
         fprintf(fi, "i = %d ", i);
         for (j = 1; j <= slength; j++)
         {
         fprintf(fi, "%d",bit_val[i][j]);
         if (j==slength) fprintf(fi, "\n");
         }
         } */
    nbold=nb;
    neold=ne;
    nemutmin = ne;
}

void mutminval(void)
{
    /* This part of program is to perform mutation of the maximum
       fitness chromesome */
    nb=neold+1;
    ne=neold+slength;
    nbmutmv = nb;

    for (i=nb; i<=ne; i++)
         {
          for (j=1; j<=slength; j++)
          {
           bit_val[i][j]=minparent[j];
          }
         }
```

```
for (i=nb; i<=ne; i++)
    {
     it2_rand=rand();
     result3=div(it2_rand,3);
     n=result3.rem+1;n=2;
     int_rand=rand();
     result=ldiv((long)int_rand,(long)(slength-n-1));
     p=(int)result.rem+1;
     for (j=i-neold; j<=i-neold+n; j++)
     {
      if (bit_val[i][j] == 0)
         bit_val[i][j]=1;
      else
         bit_val[i][j]=0;
     }
    }
nbold=nb;
neold=ne;
nb=neold+1;
ne=neold+slength;

for (i=nb; i<=ne; i++)
    {
     for (j=1; j<=slength; j++)
     {
      bit_val[i][j]=minparent[j];
     }
    }

tt=0;
for (k=1; k<=ngene; k++)
    {
     tt=tt+nbit[k-1];
     for (i=nb+tt; i<=nb+tt+nbit[k]-1; i++)
     {
      for (j=tt+1; j<=i-neold; j++)
          {
           if (bit_val[i][j] == 0)
          bit_val[i][j]=1;
           else
          bit_val[i][j]=0;
```

```
            }
          }
        }

    /*fprintf(fi, "---------mutminval------\n");
    for (i=nb; i<=ne; i++){
        fprintf(fi, "i = %d ", i);
        for (j = 1; j <= slength; j++)
        {
        fprintf(fi, "%d", bit_val[i][j]);
        if (j==slength) fprintf(fi, "\n");
        }
        } */
    nbold=nb;
    neold=ne;
    nemutmv = ne;
}


void dlmut(void )
{

    int p = 1, j1;

/**************************************
        FIRST GENE
**************************************/

    for ( i = 1; i <= ngene; i++)
    p = p * nbit[i];

    nb = neold + 1;
    ne = neold + p;
    nbdlmut = nb;

    for ( i = nb; i <= ne; i++ )
    for ( j = 1; j <= slength; j++)
        bit_val[i][j] = minparent[j];

    i = nb;
```

```
    while ( i <= ne ) {
    for ( j = 1; j <= nbit[1]; j++){
        tt = 0;
        while ( tt < nbit[2] ){

            if ( bit_val[i][j] == 0 )
            bit_val[i][j] = 1;
            else
            bit_val[i][j] = 0;

            for ( j1 = nbit[1]+1; j1 <= nbit[1]+1+tt; j1++)
            if ( bit_val[i][j1] == 0 )
                bit_val[i][j1] = 1;
            else
                bit_val[i][j1] = 0;
            tt++;
            i++;
        }
    }
    }
    nbold=nb;
    neold=ne;

/************************************************
        SECOND GENE
************************************************/
    nb = neold + 1;
    ne = neold + p;

    for ( i = nb; i <= ne; i++ )
    for ( j = 1; j <= slength; j++)
        bit_val[i][j] = minparent[j];

    i = nb;
    while ( i <= ne ) {
    for ( j = 1 + nbit[2]; j <= nbit[2]+nbit[1]; j++){
        tt = 0;
        while ( tt < nbit[1] ){

            if ( bit_val[i][j] == 0 )
            bit_val[i][j] = 1;
```

```
            else
            bit_val[i][j] = 0;

            for ( j1 = 1; j1 <= 1+tt; j1++)
            if ( bit_val[i][j1] == 0 )
                bit_val[i][j1] = 1;
            else
                bit_val[i][j1] = 0;
            tt++;
            i++;
        }
    }
    }

    nbold=nb;
    neold=ne;
    nedlmut = ne;
}


void dlmut_twice(void)
{
    int j1;

    nb = nbold + 1;
    nbdlmut2 = nb;
    ne = nb + nbit[1]*nbit[2];
    nedlmut2 = ne;

    for ( i = nb; i <= ne; i++)
    for ( j = 1; j <= slength; j++)
        bit_val[i][j] = minparent[j];

    i = nb;
    for ( j = 1; j <= nbit[1]; j++){
    for (tt = nbit[1] + 1; tt <= slength; tt++){
        for ( j1 = nbit[i] + 1; j1 <= slength; j1++){
        if ( bit_val[i][j] == 1)
            bit_val[i][j] = 0;
        else
            bit_val[i][j] = 1;
```

```
            if ( tt != j1 )
                if ( bit_val[i][j1] == 1)
                bit_val[i][j1] = 0;
                else
                bit_val[i][j1] = 1;
            }
            i++;
        }
        }
        neold = ne;
        nbold = nb;
}


void crossover1(void)
{
        int  k, beg, p, q, pp;
        div_t result;

for (pp = 1; pp<10; pp++)
        {
        result = div(rand(), cross);
        q = result.rem + 1;

        result = div(rand(), slength - q - 1);
        p = result.rem + 1;

        nb = neold+1;
        i = nb;

/*  CROSSOVER OF MAX and MAX2 */

        for (beg = 1; beg <= slength - q + 1; beg++){

            k = p;

            for ( j = 1; j <= slength; j++)

                if ( j < beg || j >= beg + q)
```

```
                bit_val[i][j] = maxparent[j];

            else{

                bit_val[i][j] = maxparent2[k];
                k++;
            }

            i++;

            k = beg;
            for ( j = 1; j <= slength; j++)

                if ( j < p || j >= p + q)

                    bit_val[i][j] = maxparent2[j];

                else{

                    bit_val[i][j] = maxparent[k];
                    k++;
                }
            i++;

    }
    ne = i;

    neold = ne;
    nbold = nb;

/*  CROSSOVER OF MIN and MIN2 */

        for (beg = 1; beg <= slength - q + 1; beg++){

        k = p;

        for ( j = 1; j <= slength; j++)

            if ( j < beg || j >= beg + q)

                bit_val[i][j] = minparent[j];
```

```
            else{

                bit_val[i][j] = minparent2[k];
                k++;
            }

            i++;

            k = beg;
            for ( j = 1; j <= slength; j++)

                if ( j < p || j >= p + q)

                    bit_val[i][j] = minparent2[j];

                else{

                    bit_val[i][j] = minparent[k];
                    k++;
                }
            i++;

    }
    ne = i;
}
    neold = ne;
    nbold = nb;

}



void crossover(void)
{
    /* This part of program is to perform the corssover between the maximum
       fitness chromesome and the minimum fitness chromesome */
    nb=neold+1;
    ne=neold+80;
    nbcross = nb;
```

```
for (i=nb; i<=ne; i=i+2)
    {
     itt_rand=rand();
     result2=div(itt_rand,cross);
     n=result2.rem+1;
     int_rand=rand();
     result=ldiv((long)int_rand,(long)slength-n-1);
     p=(int)result.rem+1;
     for (j = 0; j < kol; j++)
     if (p == beg[i]){
        if ((p + cbit[i]) > slength)
           p -= cbit[j];
        else
           p += cbit[j];
     }

     for (j=1; j<=slength; j++)
     {
       if (p <= j && j < p+n)
           {
            bit_val[i][j]=maxparent[j];
            bit_val[i+1][j]=maxparent2[j];
           }
        else
           {
            bit_val[i][j]=maxparent2[j];
            bit_val[i+1][j]=maxparent[j];
           }
     }
     }
nbold=nb;
neold=ne;
nb=neold+1;
ne=neold+100;

for (i=nb; i<=ne; i=i+2)
    {
     itt_rand=rand();
     result2=div(itt_rand,cross);
     n=result2.rem+1;
```

```
        int_rand=rand();
        result=ldiv((long)int_rand,(long)(slength-n-1));
        p=(int)result.rem+1;
        for (j=1; j<=slength; j++)
        {
          if (p <= j && j < p+n)
              {
               bit_val[i][j]=minparent[j];
               bit_val[i+1][j]=minparent2[j];
              }
           else
              {
               bit_val[i][j]=minparent2[j];
               bit_val[i+1][j]=minparent[j];
              }
        }
        }

      /*fprintf(fi, "--------crossover----------\n");
    for (i=nb; i<=ne; i++){
        fprintf(fi, "i = %d ", i);
        for (j = 1; j <= slength; j++)
        {
        fprintf(fi, "%d", bit_val[i][j]);
        if (j==slength) fprintf(fi, "\n");
        }
        } */

    nbold=nb;
    neold=ne;
    necross = ne;
}


void decigene(void)
{

int kb, ke;
/* This part is to converet binary to decimal for each gene */

nbit[0]=0;
```

```
    for (i=1; i<=ne; i++)
        {
        for (j=1; j<=ngene; j++)
        {
         chrof[i][j]=0.;
           /* printf("The random number:%d, modular:%f \n", chr_rand, chrof[i][j]); */
           /* fprintf(fi, "The random number:%d, modular:%f \n", chr_rand, chrof[i][j]); */
        }
        }


for (i=1; i<=ne; i++){

     kb = 1;
     for (j=1; j<=ngene; j++){

        gendec=0.0;

        kb += nbit[j-1];
        ke = nbit[j] + kb - 1;

        for (k = kb; k <= ke; k++){

            gendec=(double)bit_val[i][k] * pow((double)2.0,
                   (double)(k-kb));
            chrof[i][j] = chrof[i][j] + gendec;

        }
      }

}
    /*     printf("The number convertion is finished\n");*/
}

void comparison(void)
{
    int count, jbeg, flag = 0;
```

```
    i = 0;
    cbit[i] = 0;
    count = 0;
    dbit = 0;

    for (j = 1; j <= slength; j++){

        if (bit_val[imaxt][j] == bit_val[imax2][j]){
            dbit++;
            count++;
            if (flag == 0) jbeg = j;
            flag = 1;
        }

        else if (count >= 2){
            cbit[i] = count;
            count = 0;
            beg[i] = jbeg;
            jbeg = 0;
            i++;
            flag = 0;
        }

        else{
            jbeg = 0;
            count = 0;
            flag = 0;
        }
    }

    if ( count >= 2){
    cbit[i] = count;
    beg[i] = jbeg;
    i++;
    }
    kol = i;
}


double f1( double x, double y)
```

```
{
    double f1;
    f1 = a*(pow(x, b) - d)*(pow(y, c) - f) + g;
    return f1;
}

double  f2( double x, double y)
{
    double fun;
    fun = pow((x-pow(2.,c)),d) + pow((y-pow(2.,g)),h) - pow(2., o);
    return fun;
}

double f3(double x, double y, double p, double q)
{
    double f3;
    f3 = a*x*y*p + b*y*p*q;
    return f3;
}


double f4( double x1, double x2, double x3, double x4, double x5,
        double x6, double x7, double x8, double x9, double x10)
{
    double fun;
/*    fun =  x1*x2 - 2.*x2*x3 + 3.*x3*x4 - 4.*x4*x5
    + 5.*x5*x6 - 6.*x6*x7 + 7.*x7*x8 - 8.*x8*x9 + 9.*x9*x10; */

/*    fun = (x1 - 1.)*(x2 - 1.)*(x3 - 1.)*(x4 - 1.)
    *(x5 - 1.)*(x6 - 1.)*(x7 - 1.)*(x8 - 1.)*(x9 - 1.)*(x10 - 1.);*/

/*    fun = x1*x2*x3*x4*x5*x6*x7*x8*x9*x10; */

    fun = x1*x2 - x3*x4 + x5*x6 - x7*x8 + x10*x9 - x1*x5*x9;

/*    fprintf(fi,
      "\nf=%.1e x1=%.1e x2=%.1e x3=%.1e x4=%.1e x5=%.1e x6=%.1e x7=%.1e x8=%.1e x9=%.1e x10=%.1e\n",
      fun, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10);*/
```

```
    return fun;
}


double  f5( double x1, double x2)
{
    double fun;


     fun = 562500. - 22500. * x1 + 225.* x1*x1 + 9. * x1*x1*x1*x1
    - 750. * x1*x1 * x2 + 15625. * x2*x2;

    return fun;
}
```